A Hybrid Multi-Robot Control Architecture

THESIS

Daylond James Hooper

AFIT/GCS/ENG/08-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GCS/ENG/08-02

A Hybrid Multi-Robot Control Architecture

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Daylond James Hooper, B.S.B.M.E.

December 2007

AFIT/GCS/ENG/08-02

A Hybrid Multi-Robot Control Architecture

Daylond James Hooper, B.S.B.M.E.

Approved:

/signed/	13 December 2007
Gilbert L. Peterson, PhD (Chairman)	date
/signed/	13 December 2007
John F. Raquet, PhD (Member)	date
/signed/	13 December 2007
Maj Michael J. Veth, PhD (Member)	date
/signed/	13 December 2007
Mark E. Oxley, PhD (Member)	date

AFIT/GCS/ENG/08-02

Abstract

Multi-robot systems provide system redundancy and enhanced capability versus single robot systems. Implementations of these systems are varied, each with specific design approaches geared towards an application domain. Some traditional single robot control architectures have been expanded for multi-robot systems, but these expansions predominantly focus on the addition of communication capabilities. Both design approaches are application specific and limit the generalizability of the system. This work presents a redesign of a common single robot architecture in order to provide a more sophisticated multi-robot system. The single robot architecture chosen for application is the Three Layer Architecture (TLA). The primary strength of TLA is in the ability to perform both reactive and deliberative decision making, enabling the robot to be both sophisticated and perform well in stochastic environments. The redesign of this architecture includes incorporation of the Unified Behavior Framework (UBF) into the controller layer and an addition of a sequencer-like layer (called a Coordinator) to accommodate the multi-robot system. These combine to provide a robust, independent, and taskable individual architecture along with improved cooperation and collaboration capabilities, in turn reducing communication overhead versus many traditional approaches. This multi-robot systems architecture is demonstrated on the RoboCup Soccer Simulator showing its ability to perform well in a dynamic environment where communication constraints are high.

Acknowledgements

I would like to thank DAGSI and AFIT for the opportunity to pursue my Master's degree and Dr. Peterson for his continued guidance throughout the program. I especially thank my wife and son for their enduring patience throughout.

Daylond James Hooper

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

A Hybrid Multi-Robot Control Architecture

I. Introduction

A Multi-Robot System (MRS) is defined as a collection of robots unified by a mechanism common to each robot, such as communication or tasking. These systems are useful since they are not spatially limited to a central location as a single robot system. MRSs are typically more capable at tasks that require simultaneous execution of actions, are spatially displaced, and require teamwork. Granted, a uniquely capable or fast robot may execute these tasks, but there is an associated cost increase in providing the hardware and software to perform them. Also, the single robot solution is likely uniquely designed and suited for these tasks, so modification of the tasks requires additional expense in reconfiguring the robot.

MRSs, however, maintain simplicity of individual robot design and reconfigurability in the face of task modification to provide a spatially distributed approach to execution of tasks [20]. They are, by nature, more difficult to implement, and are less common than single robot systems. MRSs also often sacrifice individual autonomy and sophistication for improved coordination and collaboration capabilities. However, if implemented appropriately, MRSs provide retention of individual capabilities and the addition of capabilities reserved for multi-robot tasks. To provide these capabilities in a heterogenous system, a control architecture must emphasize both individual autonomy (independence) and collaboration and cooperation capabilities (coordination). Collaboration in this sense is when multiple robots work together, each on a separate subtask, to accomplish a task. This involves no synchronization since, though the overall task completion is dependent upon each robot completing their task, none of the individual robot's tasks are dependent upon the others. Cooperation is when multiple robots work together to accomplish a task or subtask. This

requires at least some degree of synchronization, as overall task progress is mutually dependent. If both of these are provided, the system has an even greater expansion of capabilities in general, especially if the system is heterogeneous. Modern MRSs tend to emphasize coordination while sacrificing independence, and most work involving independence is limited to single robot systems.

MRSs are often employed in situations where a task is too complex for a single robot, a task is impossible for a single robot to perform, or the risk of damage to a single robot is significant. MRSs are also used in hopes of speeding up task completion in applications such as construction, military operations, or localization and mapping. In construction or military operations, there is often a need to combine multiple robot types or heterogeneous systems into a cohesive group in order to complete a given task. MRSs used in these situations provide a significant benefit, since the work is distributed among the robots in the group. In all the above applications, there remains a need for individual autonomy, thus ensuring that failure of part of the group does not result in failure of the entire group. Many multi-robot control architectures are inappropriate for these applications, where both a high level of independence and a high level of cooperation are necessary. Some attempt to handle this via heavy use of communication, thus monopolizing communication channels and risking broadcasting of information to undesired parties, such as an enemy in a military engagement. There is often a tradeoff between independence and cooperation, since exclusive focus on either one leads to either a single-agent design or a highly cooperative yet mutually dependent multiagent design. Attempts to balance the two often result in either extraneous activity [10] or error propagation [20].

1.1 Research Goal

In order to address the above issues (communication overhead, independence, and cooperation), this thesis presents a control architecture designed to possess the following attributes:

- Low communication overhead: low communication needs contributes to a decreased chance of communication messages being intercepted by undesired parties and reduces bandwidth needs, in turn increasing the rate of accepting high-priority messages into the communication network. This, however, tends to require much greater sophistication of the individual robots in the group.
- Highly independent: each robot in the group is capable of making high-level decisions towards completion of a complex task, and can independently act upon these decisions (provided no cooperation is needed). Like the previous attribute, this requires greater sophistication of the robots in the group. It also commonly causes a decrease in the ability of the robots to cooperate.
- Cooperative: when cooperation is needed, the robots are able to work together to complete a task, and can individually select the tasks they cooperate on. This is not mutually exclusive with independence, but the two characteristics are relatively difficult to maintain concurrently.
- Expandable: introduction of new robots into the group is performed with low overhead, and new robots immediately contribute to a task.
- Robust: failure of individual robots or a number of robots does not cause failure of the entire group, and tasks that fall within the skillset of the remaining robots are completed eventually, based upon the task priority and the present allocation of the tasks.
- Extensible: addition of new capabilities to any number of robots from the group does not require reconfiguration of the other members of the group, and the architecture aspects contained within individual robots are sufficiently modular to enable rapid integration of these new capabilities.

With these properties, the architecture consists of both a multi-robot architecture and a single robot architecture, so that it is capable even as a single robot system. This minimizes the need to reprogram for specific tasks, regardless of their nature. An architecture that possesses these properties contributes a greater degree of auton-

omy and sophistication to an MRS. Chapter III provides a description of the Hybrid Architecture for Multiple Robots (HAMR), an architecture developed to deliver these properties. The key advancements of this architecture are that it provides advanced coordination capabilities through an emphasis on individual autonomy, contributes to this coordination with low communication requirements, provides a taskable system for all associated robots in the collective, possesses a straightforward mechanism for modifying the collective size, and enables mutual independence for all the robots in the collective.

1.2 Sponsor

This research is part of the Cooperative Autonomous Navigation and Intelligent Sensing (CANIS) project sponsored by the Air Force Office of Scientific Research (AFOSR). The associated autonomous navigation aspects for CANIS include the need for each agent to operate independently and also perform well in a group, in turn requiring an architecture with the properties described in the previous section.

1.3 Assumptions

In order to maintain a fairly accurate picture of the world, sensor signal-to-noise ratios are assumed sufficient to generate useable data. Real-time processing is also assumed, since the robots in the collective must interact with the world on a timely basis. It is generally assumed that the state representation is consistent, i.e., the data contained in the state is not self-contradictory. The expected domain contains a medium that enables the ability to transfer information between agents at a reasonable distance and in reasonable time. Communication constraints require that the agents are addressable, and each agent can broadcast information. Domain restrictions include an environment with a minimum spatial area such that all agents can both fit in the environment and perform work within the same.

1.4 Thesis Organization

This thesis is organized as follows: first, an examination of the requirements for a multi-robot control architecture is presented in Chapter II. Chapter III presents the Hybrid Architecture for Multiple Robots (HAMR), developed to fulfill the requirements from Chapter II. A description of the incorporation of the architecture into the simulation environment, the RoboCup Soccer Simulator, is provided in Chapter IV. Chapter V discusses the resulting architecture, provides a comparison of its performance to a similar architecture, and examines the factors contributing to the results. Chapter VI provides the research conclusions, a discussion of the architecture's contributions, and presents recommendations for future work. Finally, the Appendix holds Unified Modeling Language(UML) diagrams for the architecture's primary components.

II. Architecture Review

Any Multi-Robot System (MRS), regardless of the degree of centralization of the system, requires the presence of at least some aspect of a control architecture on each system member. Fully decentralized systems are the most complex in this regard, since the presence of the entire architecture is required on every member. These systems have a requirement for individual autonomy to maintain robustness to failure, yet must also maintain sophisticated coordination capabilities. Thus, these systems are analyzed via two approaches. The first approach examines single robot aspects of the architecture, and is followed by consideration of the multi-robot requirements. Therefore, the requirements of the final architecture are extracted by examining both the single robot and multirobot architectural components that are included in the final system.

To provide these examinations, this chapter has been broken into four sections. The first section introduces MRSs and discusses some of the necessary considerations when building a software architecture. This is followed by an examination of architecture composition for single robots and determination of the associated architecture components. Section 2.4 describes multirobot systems and examines why current architectures are inadequate for many military domains. The final section discusses some additional considerations that are addressed in the final architecture selection.

2.1 Introduction to Multirobot Systems (MRS)

MRSs have the potential to provide significant advantages over single robot systems in that they enable coordination to complete tasks that the robots individually are not otherwise able to perform, or they can enable more rapid completion of other tasks. One purpose of MRSs is to create more robust systems by taking advantage of redundancy [6] and multitasking. The robustness of the MRS generally depends upon contributions of each robot or agent[1] in the system, so the more each robot contributes to the system, the more robust, effective, and reliable the system can be.

[1]The term "agent" is used interchangeably with robot throughout this document.

A high measure of reliability and robustness is of crucial importance in military applications, since it is important to minimize unit loss and ensure that tasks are performed. Therefore, the focus of multiagent systems for this application is in maximizing the contribution of each agent to the system. This focus is broken down into two categories: 1) the robot's capability to contribute via cooperation with other robots, and 2) the robot's capability to contribute via individual actions and independent tasks. A software architecture is selected that emphasizes both of these. An architecture focusing exclusively on independent tasks does not cooperate, since there is no mechanism for coordination, in turn causing extraneous activity [10]. An architecture focusing exclusively on cooperation tends not to have robust independent capabilities, such as many swarm-style architectures [20]. This tends to cause a cascading effect, where a swarm leader can make a poor decision and it is propagated back to other system members. The cooperation and independence categories dictate that the final architecture must unify a capable single agent architecture and a robust multiagent architecture in order to capture both aspects.

2.2 Single Robot Architectures

Single robot architectures (SRAs) are typically distributed on a spectrum that spans from purely deliberative to purely reactive control [3]. Purely deliberative control architectures perform significant computations using a highly symbolic domain description, which provides sophisticated decision making but requires significant processing resources and times. The symbolic state representation present in most architectures of this nature provides an internal model that a system architecture examines in order to aid decision making [33]. Reactive control architectures, conversely, perform very little computation and are typically highly reflexive, with little or no symbolic state representation. It is assumed that reactive architectures always make a timely decision.

Some of the earliest robots, such as Shakey [45], used deliberative mechanisms. The primary benefit of deliberative architectures is that they provide high-level rea-

soning that contributes to goal-based tasking, but when computation time exceeds the time for environmental change, decisions made by deliberative architectures become obsolete before they are employed. Reactive architectures address this by possessing rapid execution of behaviors in response to environmental stimuli [11]. These architectures provide very rapid response time, but since they are representation-free, they can make the same poor decision repeatedly and fail to handle more complex tasks in general. In order to overcome the drawbacks of architectures that are designed as either reactive or deliberative (but not both), hybrid architectures have been developed that make use of the best features of both approaches. Among these are AuRA [3], 3T [9], SSS [17], ATLANTIS [33], SAPHIRA [32], TCA [49], OpenR [21], MIRO [52], DTRC [44], CLARAty [54], and Remote Agent [37]. All of these architectures contain specific modules, layers, or subsystems designed to enable and mix both deliberative planning and reactive execution. Most hybrid architectures generalize to a basic system containing three layers [33]. Therefore, a review of these architectures is provided to indicate the generalization of each to a three-layered approach. The terms used for the generalized layers follow those used by Gat [33], which consists of Controller, Sequencer, and Deliberator layers. The Controller layer handles the reactive component, tying responses directly to environmental stimuli. The Deliberator handles the deliberative component, which performs high level processing by generating plans and decomposing tasks. The Sequencer ties the two together, activating behavior sets as necessary to complete a task. The following subsections provide a brief overview of several single agent architectures, and describes their generalization to the three layer paradigm.

2.2.1 AuRA. The AuRA architecture [3] has two primary components: a hierarchical component and a reactive component, shown in Figure 2.1. The reactive component contains the motor and sensory controller, and the hierarchical component contains a mission planner, a spatial reasoner, and a plan sequencer. The spatial reasoner and the mission planner functionality is contained within most approaches as a

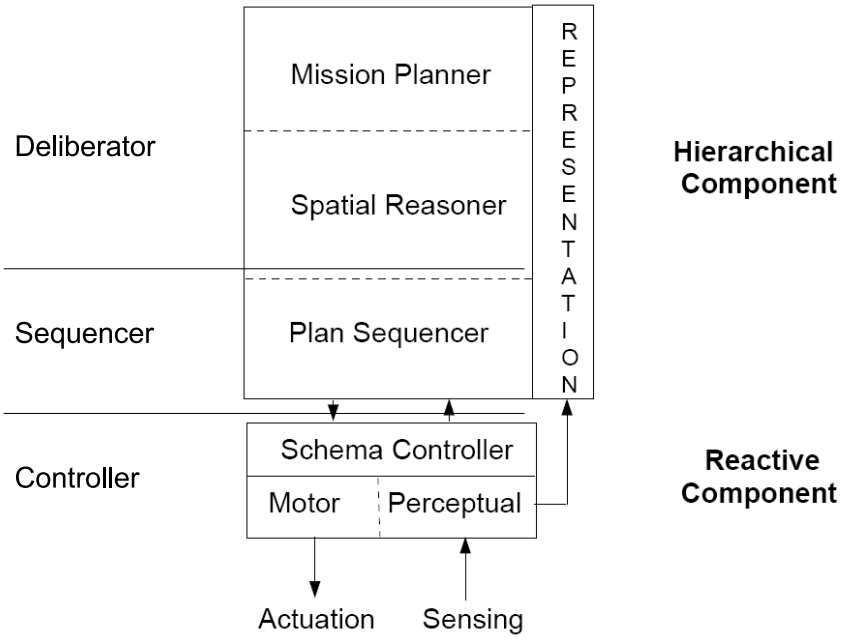

Figure 2.1: The AuRA Architecture. The Hierarchical and Reactive components are the primary portion of this architecture, where the Hierarchical component's mission planner and spatial reasoner act as the Deliberator, it's plan sequencer is the Sequencer, and the reactive component is the Controller [3].

Deliberator. This provides for the classification of the Sequencer within the plan Sequencer of the hierarchical component. The reactive component, finally, contains the Controller functionality. This results in simply a three-layered architecture consisting of a Deliberator, Sequencer, and Controller.

2.2.2 3T. The 3T architecture [9] consists of a planner, sequencer, and a skills layer tied into some external tools and an interaction layer. The skills layer interacts with the world, and represents roughly the same functionality as the Controller layer. The planner and sequencer perform similarly to the Deliberator and Sequencer, respectively. There is also an Interaction layer that functions merely as a means of enabling goal input and updates and output of the generated plan. This architecture is shown in Figure 2.2.

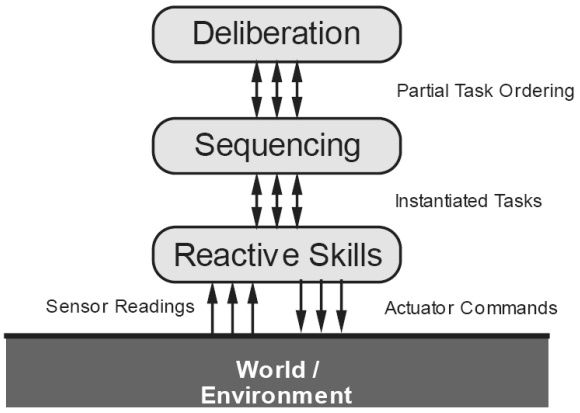

Figure 2.2: The 3T Architecture. The Planner, Sequencer, and Skills layers are equivalent to the Deliberator, Sequencer, and Controller layers [9].

2.2.3 SSS. Shown in Figure 2.3, SSS is an acronym for servo, subsumption, symbolic [17]. This architecture is so named because it combines aspects of different types of architectures. For the servo layer, it receives sensor data and outputs data to the actuators, thus acting as the Controller layer. The subsumption layer turns off or on the behaviors as necessary, acting as a behavior selection layer or Sequencer. The symbolic layer is a low level planner that basically enumerates the actions to take when events occur, thus performing the function of the Deliberator.

2.2.4 ATLANTIS. The ATLANTIS architecture [33] also has three layers: control, sequencing, and deliberative. The control layer physically performs the behaviors, the sequencing layer turns on or off behaviors, and the deliberative layer performs planning. All three of these layers follow the indicated terminology. A simplified version of this architecture is shown in Figure 2.4.

2.2.5 SAPHIRA. The SAPHIRA architecture is broken down into sections defined by focal areas: sensor processing, actuator output, Local Perceptual Space (LPS), Procedural Reasoning System (PRS), and topological planning [32]. The sensor processing, actuator output, and LPS could all reclassify as the Controller Layer,

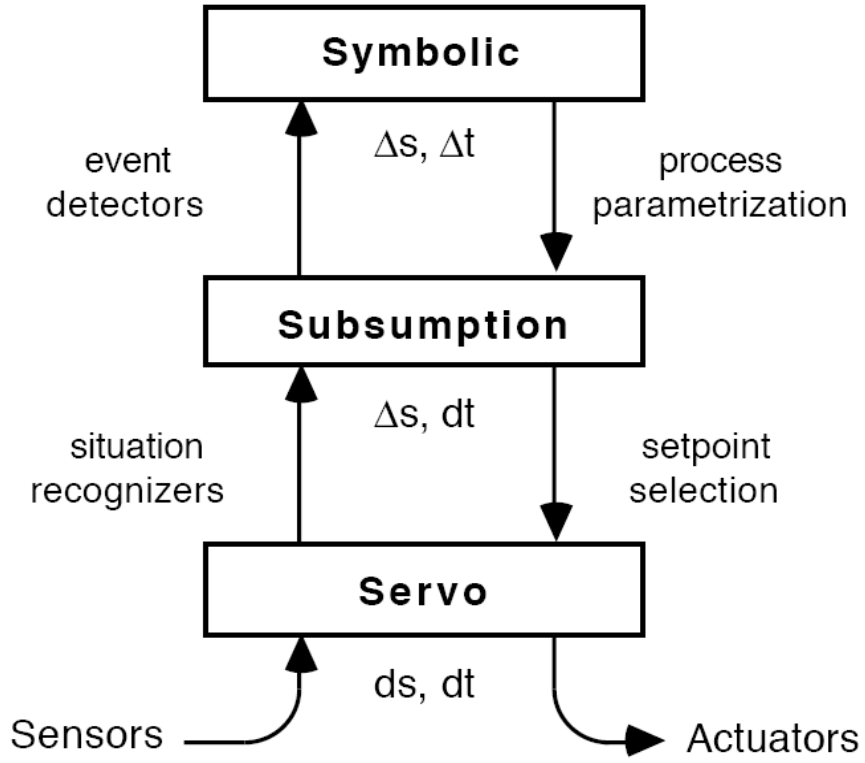

Figure 2.3: The SSS Architecture. The Servo, Subsumption, and Symbolic Layers follow the Controller, Sequencer, and Deliberator layers respectively [17].

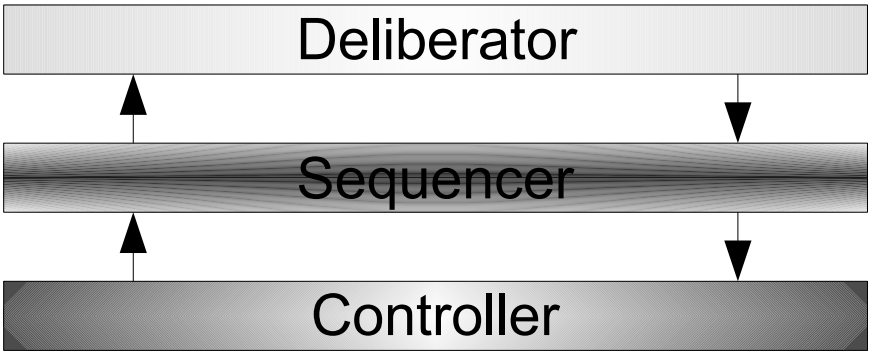

Figure 2.4: The Deliberator performs high-level computations, the Sequencer sends tasks to the Controller, and the Controller carries out the actions.

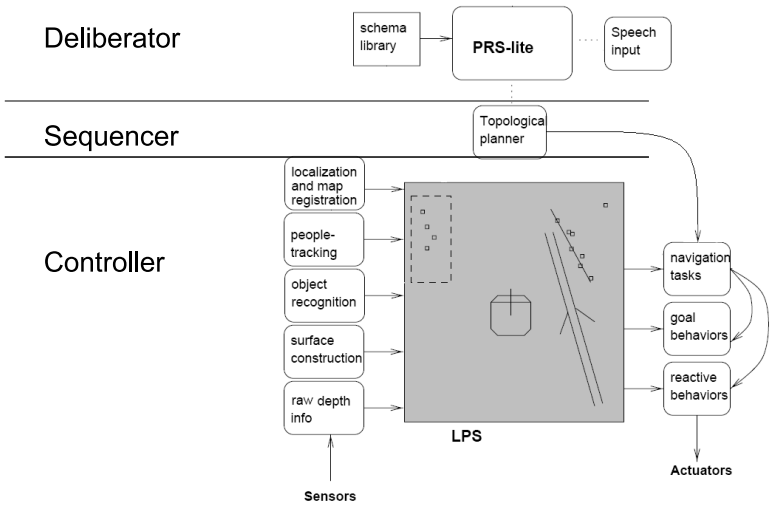

Figure 2.5: The SAPHIRA Architecture. The sensor input processing, LPS, and actuator outputs are the controller layer and the PRS and topological planner act as the Deliberator and Sequencer, respectively [32].

since the concentrations of these areas are on carrying out behaviors and processing sensor data. The PRS is an analog of the Deliberator, and the topological planner bridges the gap between the PRS and the behaviors, thereby taking on the role of the Sequencer. This architecture is shown in Figure 2.5.

2.2.6 TCA. Shown in Figure 2.6, this architecture has modules that send inputs to and get data from the Central Control [49]. This is a message-passing design, intended to also establish a hybrid control architecture. Even with the message passing approach, parallels to the three layered approach are apparent. The Central Control maintains resources and builds task trees, so it performs high-level processing in much the same way as the Deliberator. Each module controls a specific behavior, acting as the Controller Layer. The Central Control handles message passing, which is the means of communication between the Central Control and the modules. When a goal has been decomposed by the Central Control, appropriate aspects of the behaviors

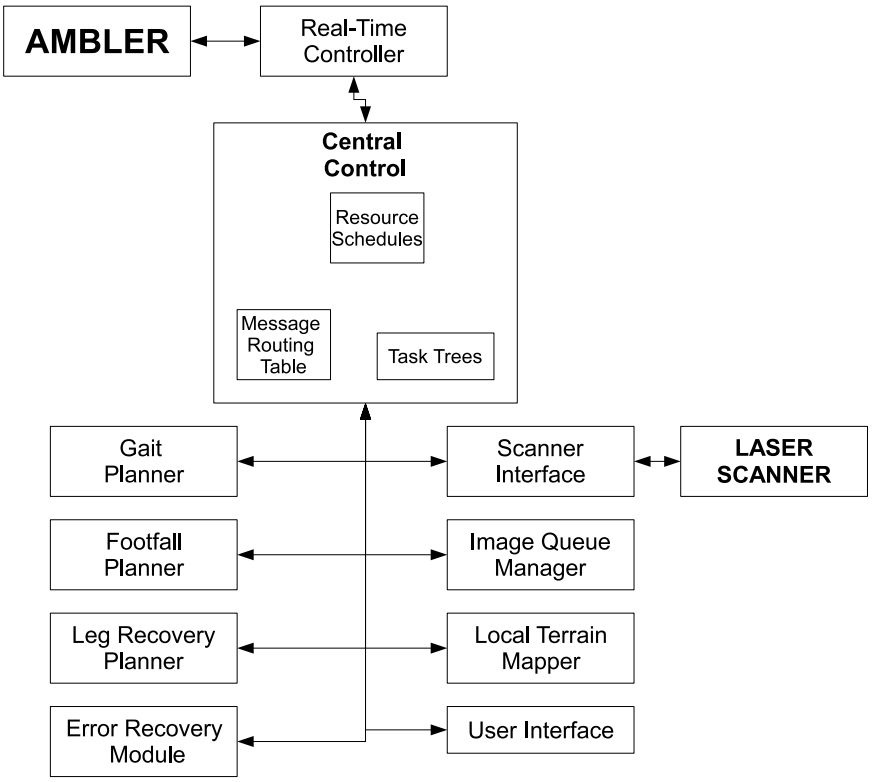

Figure 2.6: The TCA Architecture. The Central Control performs the deliberation and sequencing actions and the modules represent the Controller Layer [49].

are turned on by a *Command* message. This message passing fills the role of the Sequencer.

2.2.7 OpenR. The OpenR architecture [21] is designed to provide a framework through which users can customize a robot according to their architecture preferences. However, in order to enable the user to take either a Sense-Plan-Act approach or a Behavior Based approach, the architecture defaults to one paralleling the TLA. This is shown in Figure 2.7. From this configuration, the Target Behavior Generator acts as the Deliberator, the Action Sequence Generator acts as the Sequencer, and the Motor Command Generator acts as the Controller. These are all parts of the Application Layer design in OpenR.

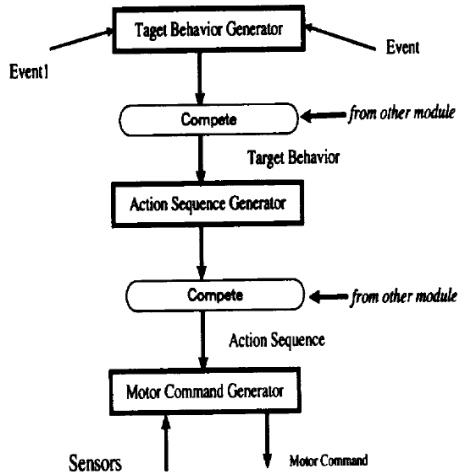

Figure 2.7: The OpenR Architecture. The Target Behavior Generator acts as the Deliberator, the Action Sequence Generator acts as the Sequencer, and the Motor Command Generator acts as the Controller. [21].

2.2.8 MIRO. The MIRO architecture, shown in Figure 2.8, is geared towards middleware, providing transitions from behaviors to specific hardware component activation [52]. This approach is composed mostly of layers of hardware abstraction. The Application and Miro Class Framework layers are hardware independent, so the functionality of the Controller is captured by the lower two layers and the behavior definitions in the Miro Class Framework. The Miro Class Framework contains within it the ability to generate a variant of the TLA, since, according to [52], "Available functionality includes a behavior engine, which permits dynamic activation, enabling and disabling of sets of behaviors and arbitrators...", allowing the Sequencer definition to also fall in this area. The application layer, then, allows development of the Deliberator. Though not broken up into three layers, this architecture still provides a hybrid deliberative and reactive architecture.

2.2.9 DTRC. The DTRC architecture shown in Figure 2.9 makes use of a modification of Graphplan [8] called DT-Graphplan, which allows for stochastic actions and probabilistic propositions [44]. This layer passes a plan to the Execution

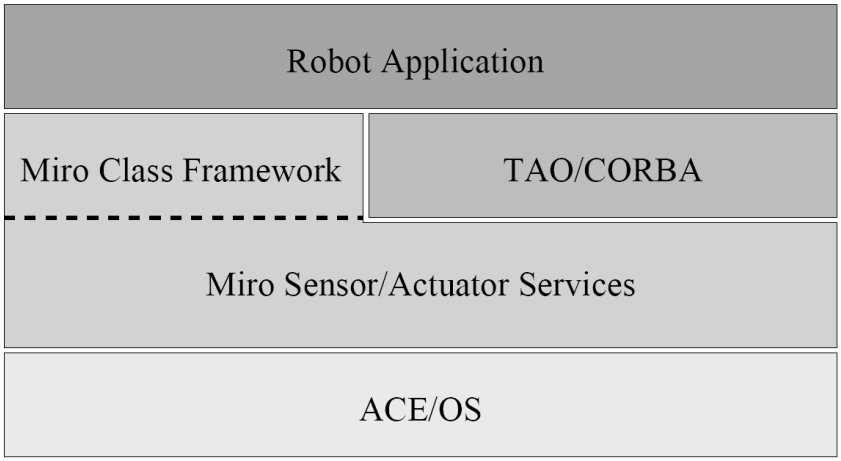

Figure 2.8: The MIRO architecture. The application and Miro Class Framework layers encompass the Deliberator and Sequencer layers of the TLA, with the Miro Sensor/Actuator services and the behavior definitions in the Miro Class Framework containing the Controller. [52].

Monitor, which in turn passes a skill activation to the Skills layer. This mapping is straightforward, as the DT-Graphplan layer behaves as the Deliberator, the Execution Monitor is the Sequencer, and the Robot Skills layer functions like the Controller.

2.2.10 CLARAty. Shown in Figure 2.10, CLARAty (Coupled Layer Autonomous Robot Architecture) consolidates the Sequencer and Deliberator Layers into a single layer in an attempt to provide Deliberative planning mechanisms with access to the system functionality (the Controller Layer) [54]. This approach, however, is still easily differentiated into the three layers, with the Planner aspects of the Decision layer serving as the Deliberator, the Executive aspects acting as the Sequencer, and the Functional layer filling the role of the Controller.

2.2.11 Remote Agent. The Remote Agent Architecture [37], shown in Figure 2.11, follows a design approach that encompasses the Deliberative and Sequencing functions with a single Remote Agent Module. The external tools, with exception of the Planning Experts, form the Controller basis. The architecture is designed to be

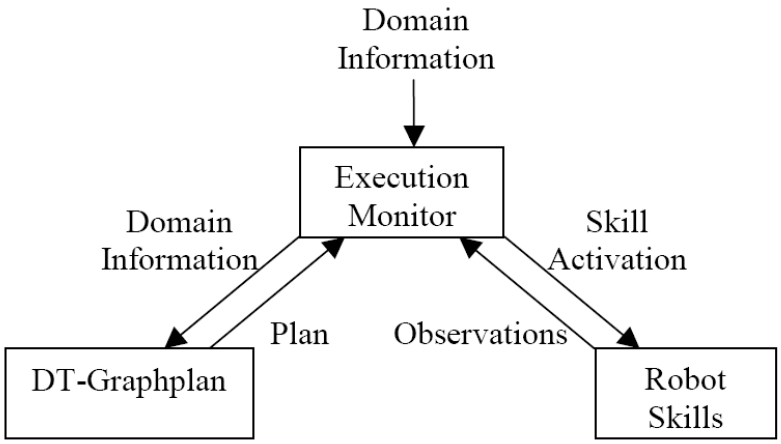

Figure 2.9: The DTRC architecture. The DT-Graphplan layer maps to the Deliberator, the Execution Monitor acts as the Sequencer, and the Robot Skills layer contains the functionality of the Controller [44].

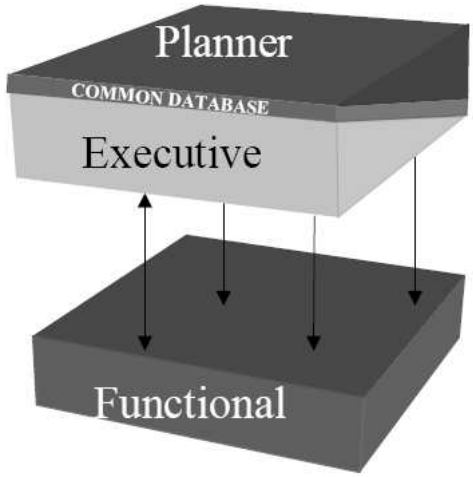

Figure 2.10: In CLARAty, the Deliberator and the Sequencer are combined into the single Decision layer and the Functional layer contains the functionality of the Controller [54].

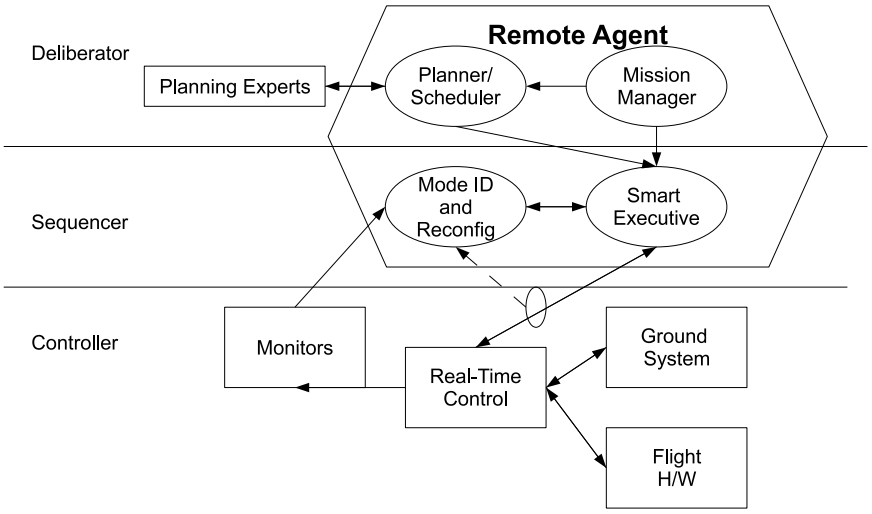

Figure 2.11: The Remote Agent Architecture contains the Deliberative and Sequencing aspects, with the Controller provided by external tools [37].

embedded in flight control software for spacecraft and fulfill necessary operations in a timely manner.

The similarity of these architectures in light of the fact that they were developed independently attests to their stability. This is still an active research area, but most modern approaches reflect design variations such as integration of additional tools, alternate solver methods, or new applications of the TLA style [36, 39, 51, 53]. For a more generalized language set with regards to three layer architectures, the terminology used throughout the rest of the document to describe the layers reflect that used by Gat [33], who calls the layers the Deliberator, the Sequencer, and the Controller. The general hierarchy of this architecture with appropriate terminology is shown in Figure 2.4. The key advantage of this architecture and its variants is its modularity: the Deliberator is completely platform independent, as it has no direct interaction with the hardware-dependent Controller. The Sequencer is also platform independent to an extent that only references to the Controller's behaviors are needed, since the Controller handles the behavior structures itself. TLAs, then, have two major contributions to robotic control: their modularity and their capabilities for relatively platform-independent structuring.

With this structure, robots are able to have both advanced deliberative capabilities and rapidly responding reactive capabilities. This architecture successfully fulfills the second category from Section 2.1, enabling both sophistication and the ability to perform well in stochastic environments. The deliberative and reactive hybrid capabilities are ideal requirements for the final architecture, and will be referred to as the SRA requirements.

2.3 Multi-Robot Systems

MRS's are expansive in terms of their design space [6] and their associated architectures are many and widely varied, since they must address many more factors than Single Robot Architectures (SRAs). These factors include incorporating design decisions regarding the nature of the collective in conjunction with consideration of the SRA aspects. The next three sections address these factors by examining two taxonomies. These are discussed with regards to the performance requirements for the final MRS architecture. Furthermore, using these taxonomies, the specific architectural construction and design approaches applied by the designers of various architectures are better understood.

2.3.1 Cao's Taxonomy. There are two taxonomies that classify MRSs. First is a taxonomy developed by Cao [14], which breaks the architecture into four categories:

1. Centralization/Decentralization. Centralization is characterized by a presence of a single control agent. MRSs without single control agents are considered decentralized.
2. Differentiation. MRS are considered homogeneous if capabilities are identical from robot to robot, heterogeneous otherwise.
3. Communication Structures. Robots can interact in any combination of three ways. First is via the environment, where environmental modifications (e.g.

stigmergy) are the only means of communication. Second is interaction via sensing, where there is still no explicit communication but the ability to recognize objects provides limited information for modeling of other robots. Third is interaction via communication, which provides the most information for modeling of other robots but adds complexity.

4. Modeling of other agents. Appropriate modeling provides the ability to reduce communication overhead and can lead to more effective cooperation.

The first category is a continuous classification, allowing a system classification to include fully centralized, fully decentralized, or somewhere in between. A fully centralized system provides system simplicity, since only one agent must be intelligent. This agent then tasks all other agents. However, this does have its drawbacks: the single point of control also creates a single point of failure. Also, the centralized system is sensitive to communication signal loss. A fully decentralized system eliminates these drawbacks, but adds the problems of requiring both a number of sophisticated agents and advanced coordination mechanisms, significantly increasing the system complexity. Systems that fall in between these two maintain some simplicity while reducing some of the communication overhead, but also face, to an extent, the issues of both centralized and decentralized architectures.

The second category, Differentiation, is a classification based upon agent types in the system. If all the agents are identical, it provides a comparatively easy mechanism for improving the system, since any capability added to a single agent is easily added to all others. However, if there is a task that the group must complete and any one agent lacks the ability to perform it, then, since all agents are the same, no agents have the capability and the task remains uncompleted. Conversely, heterogeneous systems can enable task completion even though certain group members cannot perform perform the task, but any improvement of the system is incremental since the improvement is only applied to a subset of the group.

The Communication Structures category has three classification values, but any combination of these can describe a system. The simplest structure is with environmental communication. In this, the communication is very low-level, and information on the status of other group members is not available. However, it is very low overhead. The second structure is via sensing, which still maintains a low overhead (assuming short sensor processing times) and provides some limited information on the status of other group members, such as orientation and velocity. The final and most advanced communication structure is via communication. This structure expands access to information about other agents, depending upon the degree of communication bandwidth available. Many MRSs contain at least the first two communication classifications.

The final category in Cao's taxonomy is Modeling of other agents. This category is widely varied. This begins at an implementation as simple as having no modeling, where other agents are recognized simply as environmental entities. This is simple from a developmental perspective, but requires high communication overhead. The other end of the modeling is sophisticated to a point where no communication is required, except to fix errors. This requires extensive modeling, and increases the time required to add agents to the group, since each group member must have the models built internally. It also requires expanded processing capability relative to no modeling, since model reasoning is comparatively time consuming.

2.3.2 Dudek's Taxonomy. An alternate taxonomy presented by Dudek, et al. [19] establishes a common language for the description of groups. His taxonomy has seven axes, and the classification values are described by specific terminology. The axes and classification values are shown below.

1. Size of the collective: SIZE-ALONE: one robot, SIZE-PAIR: two robots, SIZE-LIM: Multiple robots with some limit relative to the environment, SIZE-INF: multiple robots with no limit relative to the environment.

2. Communication Range:COM-NONE: no direct communication between robots, COM-NEAR: robots can only communicate with other robots that are sufficiently nearby, COM-INF: robots can communicate with any other robot.
3. Communication Topology: TOP-BROAD: robots broadcast communications to all others, TOP-ADD: robots communicate by name or address, TOP-TREE: robots can only communicate to other robots according to constraints imposed by a tree configuration, TOP-GRAPH: robots can only communicate with other robots they're linked to, according to a graph Abstract Data Type (ADT).
4. Communication Bandwidth: BAND-INF: infinite bandwidth is available, so the communication is free, BAND-MOTION: communication is roughly the same cost as moving, BAND-LOW: very high communication costs, BAND-ZERO: no communication between robots.
5. Collective Reconfigurability: ARR-STATIC: the relative spatial positioning of agents in a system does not change, ARR-COM: the robots can rearrange according to communication and sensing variations, ARR-DYN: the relationship can change arbitrarily.
6. Processing Ability: PROC-SUM: the robot acts as a non-linear summation unit (typically too simple for a robot), PROC-FSA: the robot behaves as a finite state automaton, PROC-PDA: the robot acts as a push-down automaton, PROC-TME: the robot acts as a Turing machine equivalent.
7. Collective Composition: CMP-IDENT: all robots are identical in both hardware and software, CMP-HOM: all robots have identical hardware, CMP-HET: the group of robots are heterogeneous or not physically identical.

This taxonomy is not as useful for elucidation of the required architecture as the one presented by Cao, but it provides a further decomposition of the architectural requirements. This is primarily due to the slant of this taxonomy towards communications as opposed to architectural configuration.

2.3.3 Taxonomy Classifications. The specific requirements for the intended domain and thus the architecture itself is classified according to both of the presented taxonomies. Therefore, in order to identify these requirements, the classification according to Cao's taxonomy follows:

1. Centralization: The architecture is fully decentralized, since the system must tolerate failure and have reduced sensitivity to communication signal loss. This is a requirement, since system survivability even with unit loss is crucial in order to ensure tasks are completed. If the architecture is centralized, the risk of entire system failure is significant since there is a single point of failure. Even architectures that are only somewhat centralized run this risk, since the loss of a central unit of the architecture may cause significant capability reduction. Thus, the architecture is decentralized in order to minimize system capability reduction due to unit loss. This is especially important in military applications, where system survivability and continued system productivity are more important than individual survivability. Naturally, this requires greater sophistication on behalf of each agent in the collective versus centralized systems.
2. Heterogeneity: This system may potentially have robots of different hardware configurations, so the architecture must allow this heterogeneity. If the architecture cannot effectively control a heterogeneous group, the collective is better represented as multiple groups and each group has a separate tasking structure. In order to divide the tasks among the different groups, a central solver is required, thus increasing the measure of centralization in the architecture. This also extends to military applications. It is more effective for a single mixed unit to achieve an objective than an amalgamation of multiple units. Heterogeneous units are more difficult to implement, since each diverse agent requires specialized behaviors.
3. Communication Structures: For the sake of better task coordination and a more global knowledge base, communication exists through both sensing (observation

of other agent's tasks and positions) and communication (passing messages between agents). This allows the robots to better transfer information, which allows more information to become general knowledge. They also coordinate better, since the information about task assignments and needs are transferred. This requires greater capability on behalf of the agents, since the communication hardware and sensor processing capability is required.

4. Modeling of other agents: In order to avoid task repetition or multiple assignment, the modeling of other agents must exist. This reduces overhead when coordinating on a task and updating state representations. This need not extend to the knowledge of each other's capabilities, rather, it needs to represent the existence of the other agents and their tasking. This also requires sophistication on behalf of the agents, since model reasoning is relatively expensive.

These classifications provide information that contribute to the final architecture's ideal characteristics. Therefore, these requirements are referred to as Requirements Set 1 (RS1).

Likewise, we also classify the architecture requirements using Dudek's taxonomy:

1. Size of the collective: SIZE-LIM. There are a limited number of robots in the group. It is impractical to consider SIZE-INF, since one cannot fully saturate the real world with robots. SIZE-ALONE eliminates the need for a multiagent architecture, and SIZE-PAIR is a strict limit on the collective size. Thus, SIZE-LIM is the best option for this category.

2. Communication Range: COM-NEAR. This reflects a real-world distance limitation on communications. COM-INF is impractical for a real-world application, since there is a limit to transmission distance for any signal. COM-NONE may occur intermittently due to certain terrain, but is not considered the strict case for this classification.

3. Communication Topology: TOP-ADD. The robots communicate with each other via addressing. TOP-BROAD is considered excessive, since it can saturate communication channels for little reason. There is no true collective hierarchy, so TOP-TREE is impractical because it artificially generates a communication hierarchy. TOP-GRAPH occurs at times with the TOP-ADD topology when considering subgroups, but the subgroups communicate via addressing, making TOP-GRAPH redundant by this configuration.

4. Communication Bandwidth: BAND-LOW. Information transmission volume during communication is minimized due to high communication cost. BAND-INF is impractical to consider for real-world applications. BAND-ZERO contradicts the communication structures classification from Cao's taxonomy. BAND-MOTION is roughly similar to BAND-LOW, but it is more appropriate to associate a high cost with communication in military applications in order to minimize message transmission volume and interception risk.

5. Collective Reconfigurability: ARR-COM. The robots can rearrange according to communication and sensing variations. ARR-STATIC is inappropriate, because it implies the assumption of a static collective size. If built for this reconfigurability, single unit failure would cause complete collective failure. ARR-DYN is also inappropriate, since there is no reason to expect arbitrary information to occur in the robots (unless it is through communication or sensing, which falls under ARR-COM).

6. Processing Ability: PROC-TME. The computation model utilized by each robot is a Turing machine equivalent. PROC-SUM is too simple of a representation for the system requirements. PROC-FSA, though applicable, is not as general as PROC-TME (since states can be represented via Turing machines) and is thus not the selected classification. PROC-PDA is inappropriate for much the same reason as the PROC-FSA classification.

7. Collective Composition: CMP-HET. The group of robots is heterogeneous. CMP-IDENT and CMP-HOM assume identical hardware in each agent. Though possible in a particular implementation, the design of the system must allow for heterogeneous units. Thus, CMP-IDENT and CMP-HOM are not sufficiently general for the intended application.

These ideal requirements are referred to as Requirements Set 2 (RS2). These requirements sets establish a basis for determining whether an architecture provides the necessary capabilities for the domain. Therefore, the next section evaluates current MRSs with consideration of RS1 and RS2, along with the SRA requirements from Section 2.2. An additional consideration regarding these architectures includes the communication strategies, methods, and techniques employed in each.

2.4 Multi-Robot Architectures

This section presents existing MRSs and evaluates them in the context of the requirements set forth in Sections 2.2 and 2.3. It also illustrates the general structure and design philosophies used in each architecture.

2.4.1 ALLIANCE. One architecture that fulfills RS1 is ALLIANCE [42], shown in Figure 2.12. However, ALLIANCE utilizes a TOP-BROAD communication topology, which causes heavy network load in times of high activity and violates the selected communication topology from RS2. This load can contribute to lost communication messages and increase the likelihood of message interception by a potential adversary. Also, since ALLIANCE makes use of behavior sets that are highly dependent upon the particular robot under consideration, the architecture is not platform-independent in its behavior selection mechanisms. Since modularity is considered the key when considering the single robot contribution, the TLA requirement discussed in section 2.2 is not fulfilled using this architecture.

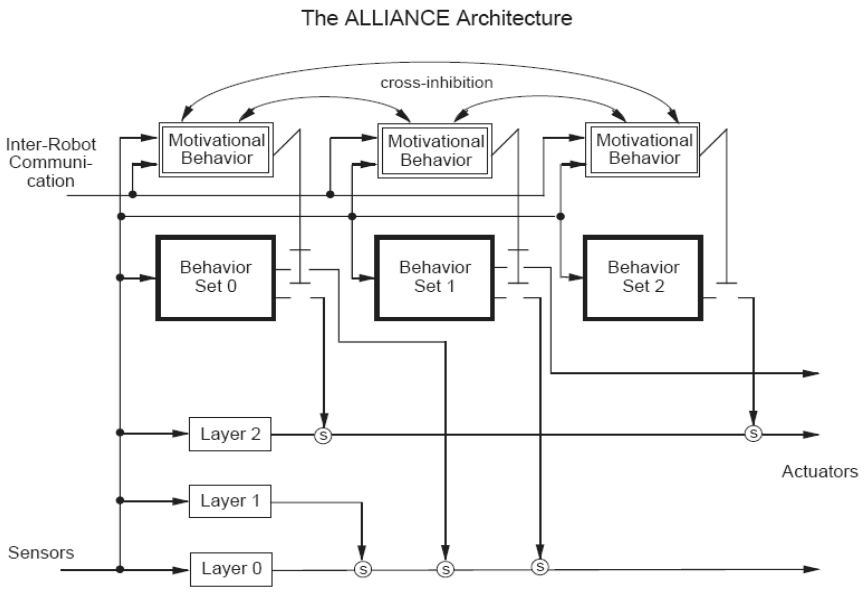

Figure 2.12: The ALLIANCE Architecture uses cross-inhibition of behaviors as its primary behavior selection mechanism [42].

2.4.2 RoboSkeleton. The RoboSkeleton Architecture [13] is shown in Figure 2.13. This architecture fulfills RS2 and the differentiation, communication structure, and agent modeling aspects of RS1. It incorporates low overhead communication from one agent to another, where the only communication based actions are changing the global leader and locating the local leader. Between the agents and the CoachAgent, however, the communication uses more bandwidth, since the strategy is communicated to all the agents. Therefore, it fails to accomplish the decentralization requirement from RS1. This architecture is built on a TLA variant, but since it contains an agent that controls the other agents, it is not fully decentralized. If the controlling agent (CoachAgent in this architecture) fails, the entire system could readily fail. This architecture, then, fails to fulfill the architecture requirements.

2.4.3 CAMPOUT. CAMPOUT [27] is an architecture that is designed for planetary exploration, outpost site preparation and maintenance, and remote science investigations. Shown in Figure 2.14, it is a behavior-based architecture with four types of behaviors. First are the primitive behaviors. Above those are composite

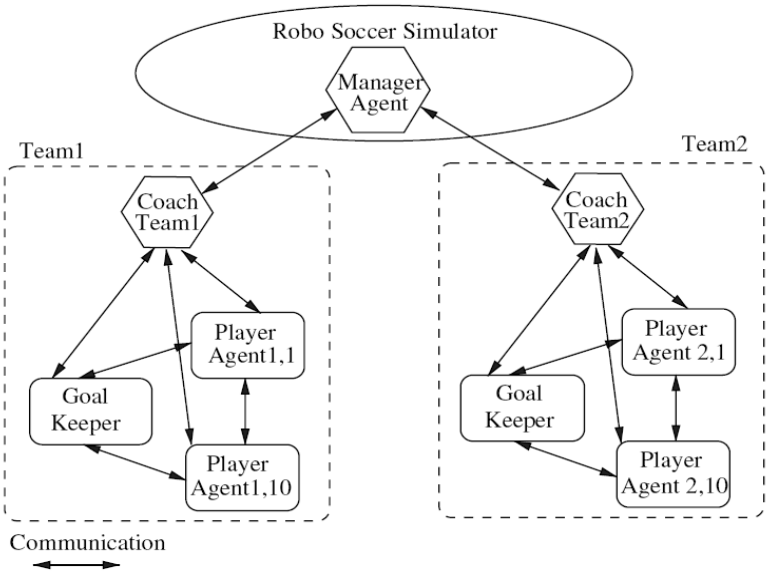

Figure 2.13: The RoboSkeleton Architecture is hierarchical and somewhat centralized [13].

behaviors, which are combinations of primitive behaviors unified via a command-fusion arbiter. Next are shadow behaviors, which mimic other robot's behaviors, and last are group behaviors, which are designed to coordinate the robots. This architecture fulfills RS1 and most of RS2, with the exception of the communication bandwidth. Since the robots can communicate at the sensor level, a small number (relative to the group size) of CAMPOUT robots can end up utilizing much or even all of the communication bandwidth, which is the source this architcture's failure to fulfill RS2. Also, this architecture is behavior-based, meaning that it is mostly reactive. Thus, the robots do not perform appropriate high level processing and the CAMPOUT Architecture does not fulfill the SRA requirements.

2.4.4 Essex Wizards '00. Another architecture is the one employed by the Essex Wizards '00 RoboCup Soccer Simulator team [26]. In this architecture, the Sensors take data from the server and they, along with the Play Mode, Parameters, and Memory modules, feed the data to the behaviors. When generated, the behaviors send motor commands to the actuators. This provides the ability to schedule

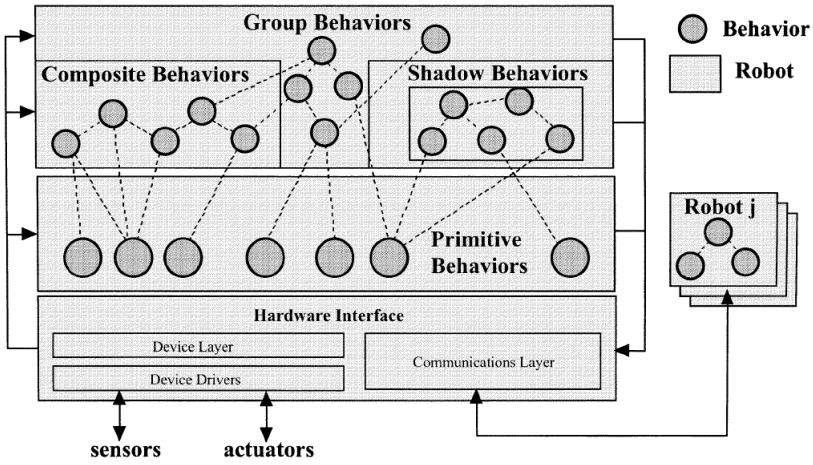

Figure 2.14: The CAMPOUT Architecture is a hierarchical, behavior-based approach. [27].

behaviors, such that all behaviors have a share of the full time available. Though there is a hierarchy of behaviors, it is contained entirely within the behaviors and neither high-level processing (deliberative tasks) nor sequencing are performed. Thus, though this architecture can fulfill both RS1 and RS2, it does not fulfill the SRA requirements. The communication in this architecture is also quite low level, since, for the most part, it is performed indirectly via sensing. This does not preclude true communication with this architecture, but represents an additional concern regarding the potential of this architecture.

2.4.5 UM-PRS. The UM-PRS [35] architecture is based upon the Procedural Reasoning System (PRS) developed by Georgeff [22], and is shown in Figure 2.16. The data flow in UM-PRS runs from the Environment through Sensors/Receivers into the Database. From there, the Interpreter unifies the procedure specifications from the Knowledge Areas (KA), the goals, and the Intention Structure (goal progress) to determine a plan that includes both high-level and low-level goals. The Intention Structure then activates certain goals and the Effector/Transmitter module handles the physical execution. The communication data is sent by the Transmitter and received by the Receiver. This architecture fulfills all of both RS1 and RS2. Its com-

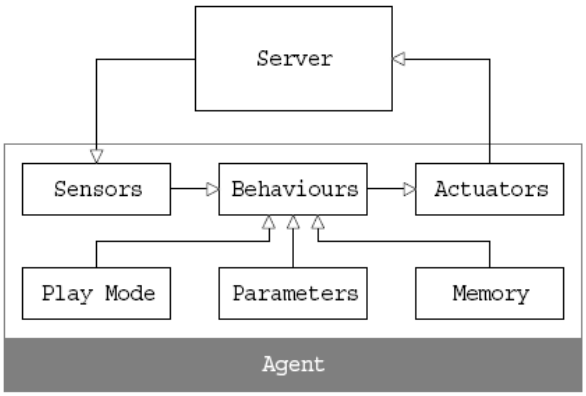

Figure 2.15: The Essex Wizards agent architecture uses a decentralized, behavior-based approach [26].

munication mechanisms minimize communication volume, since the system expects most messages to contain outdated or erroneous information by the time they are received. However, since information has to flow through the high-level processing step performed by the Interpreter at all times, this ends up being a highly Deliberative architecture. Thus, it does not fulfill the SRA requirements since it is not sufficiently reactive to operate well in dynamic environments.

2.4.6 ABBA. The ABBA [29] architecture is almost purely reactive. In fact, ABBA was intentionally designed to avoid utilizing a hybrid architecture. Also, the modeling of other agents is weak, since it is a mere recognition of the existence of additional agents. The communication follows closely to the requirements, but the tasking of the robots from the architecture is very specific and is naturally reduced in necessary communication volume. Therefore, though ABBA fulfills RS2, it is not an appropriate architecture according to the SRA requirements, since it cannot perform the higher level reasoning needed by the SRA requirements.

2.4.7 Layered Multirobot Architecture. The final multi-robot architecture considered is described by Simmons et al [48], shown in Figure 2.17. This architecture is a Layered Multirobot Architecture (LMA) based upon the 3T Architecture, so, unlike the MRSs discussed above, it fulfills the SRA requirements discussed in Section

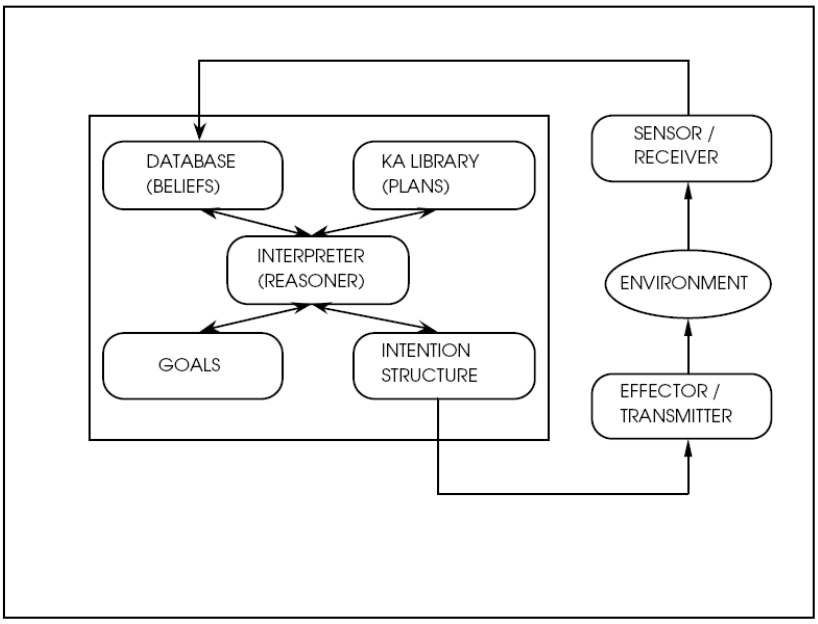

Figure 2.16: The UM-PRS architecture. This architecture requires high-level planning for any environmental change, and is thus highly Deliberative [35].

2.2. Also, it fulfills RS1, so it has the potential to fulfill the established requirements. However, one may note that the interaction from one agent to another is on all layers. This layer-to-layer cross-robot communication creates a high-bandwidth requirement for communication, thereby failing to fulfill RS2. At the deliberative level, the layer-to-layer communication is beneficial, since it enables coordination of plans. On the behavior and sequencing level, however, this is a less than ideal situation. For example, if one of the robots is damaged, it can send erratic behavior commands to the other members of the group, thereby propagating its operational limitations to the other group members. Even if undamaged and in a dynamic environment, the cross-robot behavior activation may act as an inhibitor, preventing the robots from completing a necessary task. This also takes away from the ease of expansion of the group, since each robot needs to know what all others are capable of in order to communicate on the Controller layer. Addition of new robots or new capabilities to one robot requires updating of the available behaviors on every other robot. Therefore, though this

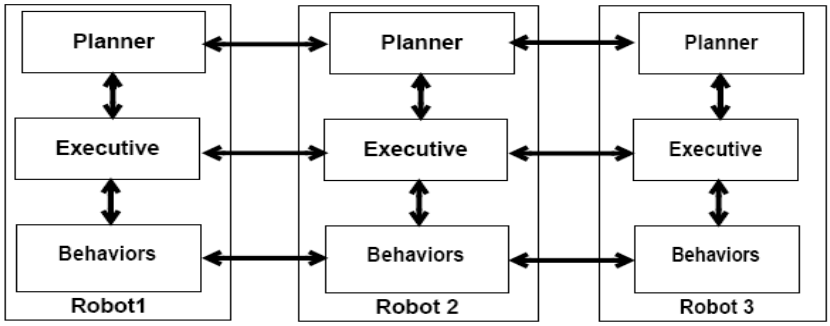

Figure 2.17: This architecture is a Layered Multirobot Architecture [48].

Table 2.1: Classification of MRSs via Cao's taxonomy. Italicized entries indicate differences between architecture and target classifications.

	Centralization	Differentiation	Com Structs	Modeling
Alliance	decentralized	heterogeneous	communication	moderate
RoboSkeleton	*centralized*	heterogeneous	communication	moderate
CAMPOUT	decentralized	heterogeneous	communication	moderate
Essex Wizards	decentralized	heterogeneous	*sensing*	moderate
UM-PRS	decentralized	heterogeneous	communication	moderate
ABBA	decentralized	heterogeneous	communication	*weak*
LMA	decentralized	heterogeneous	communication	moderate
Target	decentralized	heterogeneous	communication	moderate

architecture takes advantage of the modularity of TLAs, it is not modular in terms of robot-to-robot interactions.

Table 2.1 and Table 2.2 show each MRS and how they are characterized. From these tables, one may note that none of the presented multirobot architectures are appropriate for the architecture requirements developed via RS1, RS2, and the SRA requirements. These approaches, though capable within their targeted domain, either possess a different multirobot classification, fail to possess the single-agent architecture requirement, or are potentially sensitive to robot failure. The final architecture implemented for this domain must, then, base itself upon a robust SRA and must have reduced sensitivity to failure.

Table 2.2: Classification of MRSs via Dudek's taxonomy. Italicized entries indicate differences between architecture and target classifications.

	Size	Range	Topography	Bandwidth	Reconfigurability	Proc Ability	Composition
Alliance	LIM	NEAR	*BROAD*	*HIGH*	COM	TME	HET
RoboSkeleton	LIM	NEAR	ADD	LOW	COM	TME	HET
CAMPOUT	LIM	NEAR	ADD	LOW	COM	TME	HET
Essex Wizards	LIM	NEAR	ADD	LOW	COM	TME	HET
UM-PRS	LIM	NEAR	ADD	LOW	COM	TME	HET
ABBA	LIM	NEAR	ADD	LOW	COM	TME	HET
LMA	LIM	NEAR	ADD	*HIGH*	COM	TME	HET
Target	LIM	NEAR	ADD	LOW	COM	TME	HET

2.4.8 Related Work. MRS designs are not generally established using a common methodology or formalism, rather, they tend to be assembled from extensions of informal and undocumented expert knowledge or even trial-and-error [28]. However, there are a number of formal methodologies available. Parker [41] presents information invariants as a means to map sensori-computational systems to a mission, thus providing a mechanism for translating mission objectives into robotic resource needs and establishing a basis for the design. Balch and Arkin [5] examine the effect of communication on performance in multiagent robotic systems, which provides a basis for communication system design aspects of an MRS design. Dudek [19] and Cao [14] present taxonomies that allow MRS design characteristic extraction when examined in terms of a domain or mission objective (the formal methodology used in this thesis). Other approaches to MRS design include analysis of empirical demonstrations of MRSs, such as examination of MRS architectures. An example of this [40] presents an MRS design methodology that generates the ALLIANCE architecture through examination of the domain and may be used as a basis for other designs. Jones [28] presents a formal MRS design methodology, which focuses on controller synthesis and both macroscopic and microscopic modeling to predict task performance of the synthesized MRS. The approach he presents is not applicable for

this work, since his approach does not consider control systems using both internal state and communication, both of which are system requirements extracted from the taxonomy classifications. Kinny and Georgeff [30] present an Object-Oriented (OO) modeling technique for analysis and design of Multiagent Systems (MASs). Their OO approach presents three primary models: an Object Model, a Dynamic Model, and a Functional Model. These characterize the objects, the states and transitions, and the data flow within the system. This is presented in order to refine an MAS design in terms of the object definitions. Other approaches [55] [12] also focus on representation of agents in terms of OO models to aid design. The approach in this thesis makes use of the taxonomies to extract the MRS characteristics along with examination of MRS architectures to provide a basis from which to synthesize the appropriate MRS design. OO class diagrams are provided in Appendix A to show the agent class structuring breakdown.

2.5 Additional Considerations

Section 2.2 showed that the SRA requirements is fulfilled using current architectures. Section 2.3 showed that current multi-robot architectures are inappropriate for the particular domain under consideration, when measured against requirements developed using two taxonomies and evaluating them in terms of the SRA requirements. In continuation of these, this section addresses concerns with unification of the two. The combination of TLA with multiagent capabilities is a problem beyond a simple addition of communication capability and the ability to recognize allies. Distribution of tasks, coordination, task maintenance (methods used to ensure timely or successful task completion and avoidance of task clobbering), and survivorship (the ability of an agent or collective to survive a task or collection of tasks) are among the primary concerns of a robot collective when applied to a military application or many construction or localization/mapping applications. Therefore, there must be a way for the group to determine the appropriate agent or group of agents for a task, ensure that the tasks are being carried out in an efficient manner, and complete

tasks by certain deadlines (task management). Furthermore, each robot in the group must independently determine the task allocation (the selected distribution of tasks among multiple robots), since the system must be fully decentralized. The failure of the existing multi-robot architectures discussed above to fit well into the intended domain, then, calls for the development of a new architecture or an expansion of one that currently exists. As discussed in Section 2.3, the architecture must build upon and reflect the individual capabilities of an SRA. Therefore, we present a robust architecture which is an expansion of an SRA. The development of this architecture is discussed in Chapter III.

III. Architecture Development

This chapter presents the Hybrid Architecture for Multiple Robots (HAMR) structure. The components of HAMR reflect those of three layer architectures, with an additional layer presented for multi-robot systems to provide coordination between independent agents. This layer, the Coordinator, provides the necessary extensions to enable coordination among these agents. HAMR is designed for use in either a robot or a simulated environment, since most of the variations between the two are in the definition of the behaviors. Thus, HAMR's design is presented in a manner that is not constrained to a platform. Rather, the design presentation discusses the required functionality of each component of the architecture and describes their responsibilities, including the interchanges between the layers.

3.1 Development of the Final Architecture

HAMR is an expansion of the Three Layer Architecture (TLA) [33], shown in Figure 3.1. The lowest layer is the Controller, which contains low-level behaviors that interact closely with sensors and motors. The middle layer is the Sequencer, which translates plans from the Deliberator into a series of behaviors, in essence turning behaviors on or off, along with storing and maintaining the world state for the sake of replanning with the Deliberator. The top layer is the Deliberator, which performs high-level symbolic computations and plan development for long-term goals. The expansion comes from the addition of the Coordinator layer, described in Section 3.5. Since there are variations in previous TLA implementations [3, 9, 17, 33] with regard to the roles played by each portion of the hierarchy, a full explanation of the hierarchy and the responsibilities of each layer is included in this chapter. To provide a general approach to the architecture, specific theoretical examples are referenced throughout. These examples do not constrain HAMR to a particular domain, but serve to illustrate the construction of HAMR and its decision procedures.

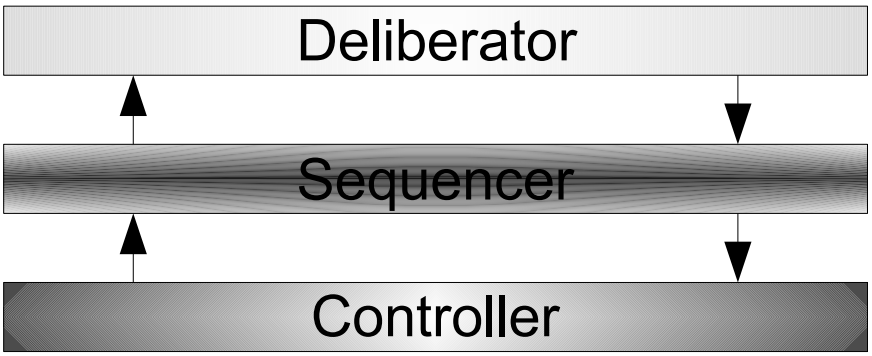

Figure 3.1: The basic Three Layer Architecture. This is the same structure shown in Figure 2.4.

3.2 Controller Structure

The Controller's specific internal structure is highly dependent upon the hardware available, but there are certain levels of abstraction available to further expand the Controller's capabilities and provide a measure of independence. The primary difference between the TLA variations discussed in Section 2.2 is where the functional separation between the layers is placed. For this domain, however, integration of specific tools enable a clean delineation between the Controller and the Sequencer. This keeps overlapping responsibilities to a minimum, since the Controller selects between and combines behaviors from the set provided by the Sequencer and never needs to generate these behavior sets. The behavior set generation is left to the Sequencer.

The Controller's responsibility falls in the area of behaviors and their physical implementation. There is still some debate as to the appropriate level of sophistication that should be encapsulated within a behavior. More specifically, a behavior can be as simple as a single motor command, or as sophisticated as an item retrieval behavior that contains obstacle avoidance and traversal. Overly sophisticated behaviors need to monitor and fulfill multiple goal states, and lead towards a more behavior-based type of architecture. Conversely, the very simple behavior leads to a more deliberative type of architecture. For the purpose of this paper, a behavior is considered to be a sophisticated function that can output multiple motor commands at the same time, yet is still simple in that it is designed to only fulfill a single goal or multiple simple

goals. This allows a selection mechanism to apply a particular behavior at a time that it deems most effective [25]. As an example of a behavior in this context, a behavior called *GoTo* may exist. When passed parameters in two-dimensional space, *GoTo*(x, y, θ) relocates the robot to (x, y, θ) relative to the robot's world frame (where x is the horizontal component, y is the vertical component, and θ is the orientation angle). The theoretical *GoTo* behaviro is more high-level than a simple motor activation because it outputs the hardware-dependent motor commands necessary to relocate the robot to (x, y, θ), without any obstacle avoidance. The hardware-dependent sensor feedback (e.g. odometry or vision sensing) updates the internal state representation, and goal completion is detected by the Sequencer. From this perspective, the *GoTo* behavior may exist on multiple types of robots with very different hardware, yet still perform the same function. For a higher level of abstraction, the behaviors are implemented as classes. Based upon an examination of the world representation (World Model), the behaviors determine some of their own parameters, such as a utility value for their behavior execution. This becomes very helpful when operating in stochastic environments, such as an obstacle-ridden course.

Consider another theoretical behavior called *Wander*, which avoids obstacles by choosing a random direction of travel whenever an obstacle is detected nearby. To apply it in the context of the previous paragraph, it may be necessary to stop execution of the *GoTo* behavior in order to circumvent an obstacle, so the *Wander* behavior then takes effect. However, instead of *Wander* subsuming *GoTo*, *GoTo* merely reduces its utility value (implying that it has less confidence that it is the best behavior to utilize in this situation), which then allows *Wander* to be selected as the best behavior. Once the obstacle is passed, the *GoTo* behavior can then increase its utility and again be selected as the best behavior. The reason for representing these behaviors as classes instead of functions is that it modularizes them, allowing each behavior to be selected via a reference instead of called within another function. The representation as classes also allows for a more hardware-independent functionality, thereby enabling multiple platforms to contain the behavior by simply calling a function in the same manner,

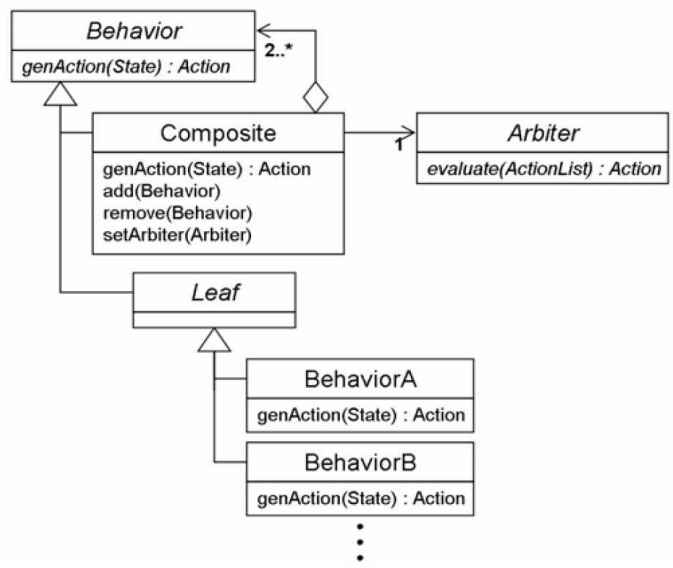

Figure 3.2: The internal structure of the UBF. Using an arbiter, a composite behavior is generated that makes use of multiple simple behaviors. [56]

regardless of the underlying processing system, motor hardware, and sensor hardware available to the robot.

Another consideration for behavior selection is not utility, but the competitive or non-competitive nature of certain behaviors. Specifically, *GoTo* and *Wander* are competitive behaviors in that only one can be selected at a time. However, moving a sensor array or manipulating a separate gripper arm is likely independent of *GoTo*. By separating the behaviors into their respective motor allocation requirements, a combination of behaviors is selected that both wanders and rotates the sensor array. This enables, in a sense, multiple behaviors to be performed at the same time as one composite behavior. The Unified Behavior Framework (UBF) is a tool for enabling both the selection of behaviors by utility and generating composite behaviors from a collection of non-competitive behaviors. Depending upon the arbitration technique selected, different environmental responses (composite behaviors) are generated. The arbitration methods range from a "winner take all" scheme to a sophisticated fusion of the component behaviors weighted by utility. They include priority-based arbitration [2] [11], command fusion [1], utility fusion [46], or a network of some combination

of these [16]. The priority-based arbitration allows behaviors with higher priorities to subsume behaviors with lower priority, and is readily provided via a priority determination within the behavior. The command fusion arbitration scheme generates a behavior that accepts aspects of all contributing behaviors and generates a mean, allowing competitive behaviors to all contribute to the final action. The utility fusion arbitration method takes into account the actuator groups used by each contributing behavior, generating a single behavior that makes use of all the contributing behaviors' actuator groups. The hierarchy of the behaviors in the UBF places simple behavior classes at the lowest level, encapsulated as a Leaf behavior class [56]. On the same level as the Leaf class is the Composite class, which contains an arbiter and some contributing behaviors. On the topmost level is a template Behavior class. All behaviors contain a genAction method, which generates the hardware commands. The UBF takes advantage of polymorphism by overriding the genAction methods of upper classes in the hierarchy. This hierarchy is shown in Figure 3.2.

The operation of the UBF consists of adding candidate behaviors with their associated weights to the Composite class then calling the genAction method for that class. The UBF then calculates, using the arbiter, the best parameters for each member motor group according to the arbitration scheme employed. The UBF also allows arbitration among Composite behaviors, so the resulting Composite behavior can consist of an extensive Composite and Leaf behavior network, with each Composite containing a specialized arbiter.

The advantage of the UBF is that it enables rapid, reactive performance of behaviors while still providing a dynamic selection process for the behaviors. The dynamic selection process allows the more time-consuming processing that takes place in the Deliberator and Sequencer layers to be avoided until sensor data indicates a requirement for a different set of candidate behaviors to be added to the Composite class. For example, given a simple task of getting to a target location (x,y,θ), the Sequencer adds *GoTo*(x,y,θ) and *Wander* to the Composite class and simply monitors the World Model. The robot, then, behaves in a reactive manner by running

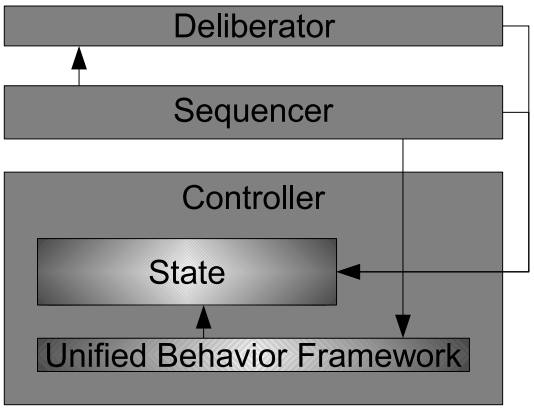

Figure 3.3: Updated version of the architecture including the UBF. The Sequencer sends candidate behaviors to the UBF for selection via arbitration.

GoTo until sensor data causes the vote of *Wander* to increase or the vote of *GoTo* to decrease to an extent that *Wander* is chosen by the arbiter, enabling the robot to wander around the obstacle. Barring any significant environmental changes, the reactive control method proves sufficient for performing this simple task. Upon task completion, the Sequencer then adds the next set of candidate behaviors to the UBF. This theoretical execution procedure captures the general flow of execution as it exists in implementation of HAMR.

The responsibility of the Controller is delineated by the top end of the UBF. It is the responsibility of the Sequencer to add candidate behaviors to the UBF, but it is the Controller's responsibility to select from those candidates through the UBF. The sensors, which are controlled via this layer, update values in the state representation, from which data is extracted by the behaviors in order to determine utility values and motor commands. The architecture as it stands with these changes is shown in Figure 3.3.

In summary, the entire functionality of the Controller is established by three core responsibilities: defining the behaviors; updating the state representation from sensor data; and physically executing the behaviors, feeding parameters to them through the state representation. The Controller also contains the State, for the simple reason that the Controller calls upon and modifies the State much more often than the other

layers. All the higher-level functionality is encapsulated in the upper layers. The modularity of the architecture makes it such that the Controller has full domain over its responsibilities, and the upper layers merely reference the state representation and trigger the candidate behaviors and arbitration schemes of the UBF for the Controller functionality.

3.3 Sequencer Structure

The responsibilities of the Sequencer are to enable and disable behaviors at the appropriate time and maintain internal state. The first of these responsibilities retrieves tasks from the Deliberator and adds the appropriate behaviors for fulfilling the tasks to the UBF. Revisiting the example in Section 3.2, the presented task is to get to a particular location and orientation. In terms of the theoretical example presented previously, the Sequencer adds all possible behaviors needed to perform the task to the UBF, which are $GoTo$(x,y,θ) and *Wander* (assuming all other behaviors in this example possess functionality unrelated to the task). They are then carried out reactively. From this example, it can be seen that the Sequencer takes a task (or subtask) from the Deliberator, determines the behaviors needed to perform it, and provides the behavior set and hierarchy to the UBF, which builds a Composite behavior from the set. In other words, the Sequencer processes tasks into candidate behaviors then provides them to the Controller layer through the UBF. Therefore, the Sequencer must have a means of determining the suitability of a behavior for a particular task. Solutions to this range in sophistication, from predicting the outcome of the behavior via decision-theoretic approaches [44], perceptual sequencing [4], or expected outcomes of actions from Reactive Action Packages (RAPs) [9], to evaluating the state representation by situation recognizers [17], and static activity schema passed down from the planner [38] [32].

The approach applied in this document assumes that the Sequencer determines suitability of a behavior by an evaluation of the state representation. For example, given a robotic soccer player with behaviors that include pass/shoot, dribble, find-

Ball, and turnNeckToBall, the Sequencer simply examines the state representation, determines that the ball is too far away to be kicked, and does not add the pass/shoot and dribble behaviors to the UBF behavior hierarchy. The findBall and turnNeck-ToBall behaviors are added and processed based upon their evaluation of the state representation and the UBF's selected arbitration method.

A secondary responsibility of the Sequencer is monitoring of the state representation for situations that require replanning. These situations include hardware failure, the "kidnapped robot" problem, changes in data, task failure, and task completion. In hardware failure, a component that is to be used for a task fails. This requires replanning to determine a method of completing the task without the component. The "kidnapped robot" problem is where the robot is relocated to a new area. Even if this area is within the map data, it occasionally requires replanning, since entire categories of behavior sets are no longer available. Changes in mapping data include finding the only known accessible path blocked. The replanning includes path planning through or around this change, beyond what the Sequencer is capable of performing. Task failure is when the robot determines that a task is impossible to be completed. There are a number of ways for the robot to determine task failure, including maintaining a deadline on task fulfillment or detecting failure of a behavior set. With the responsibility allocation here, it is the Deliberator's responsibility to establish the deadline, and it is the Sequencer's responsibility to detect behavior failure and determine if the deadline has been reached. Behavior failure is normally indicated by looping. This arises as either a short loop, such as freezing in position, or a longer loop, such as wandering around a room looking for an exit. Enabling a memory of previous states provides a mechanism through which failure detection is enabled. Replanning is then required if the Sequencer detects any of the situations, and there are two ways to perform replanning:

- Have the Sequencer regenerate the candidate behaviors. If there is appropriate change in the state representation, regeneration of the behaviors has the po-

tential to modify the actions available to the UBF and enable activation of the appropriate ones, thereby exiting the failure loop.

- Have the Deliberator regenerate the tasks. If the task decomposition at the previous run was poor, a reevaluation by the Deliberator with new state information provides a new task decomposition with alternate subtasks, and the ensuing Sequencer evaluation provides a different set of candidate behaviors.

The second method indicates the need for alerting the Deliberator for recalculation of the task decomposition. This is the third function of the Sequencer. The Sequencer monitors outcomes of behavior sets and, if the behavior set doesn't have the expected outcome within the necessary time frame, the Sequencer alerts the Deliberator which generates a new plan from the updated state and provides it back to the Sequencer for behavior set generation. For more sophisticated tasks or a series of tasks, a more high-level algorithmic approach such as Partial Order Planning is required, along with means of detecting duplicate states and ensuring progress. This alternate approach minimizes Deliberator usage, since much of the high-level solving is performed by the Sequencer.

HAMR does not have a high-level planner within the Sequencer (since the Deliberator generates appropriate task decompositions and plans), so the full responsibility of the Sequencer is captured within its three responsibilities. Once again, these are: 1) process tasks into candidate behavior functions and send them to the UBF, 2) monitor the state representation for replanning triggers, and 3) alert the Deliberator if replanning is necessary. In general, the intent of the Sequencer is to activate and deactivate behavior sets and provide more sophisticated behavior generation without very high-level processing. For a simple example of this, consider a robot tasked to retrieve an item. If the robot possesses prior knowledge of the item's location, it activates the behavior set that provides traversal to that specific target location. However, once it arrives, it finds that the item has been moved. The Sequencer observes the failure of this behavior set and activates a behavior set that provides wandering capability (allowing the robot to locally search for the object). When the

object is found, the Sequencer activates the behavior set used to pick up the item, then once again activates the target location traversal behavior set to return to the original location, all prior to the expiration time established by the Deliberator. This entire task fulfillment sequence is satisfied by a single plan from the Deliberator, and the Sequencer performs most of the active decision making needed to carry out the task.

3.4 Deliberator Structure

The Deliberator performs high-level reasoning tasks that include task decomposition, task allocation, and planning. The Deliberator is presented with a goal set that it decomposes into manageable tasks. In certain cases, the presented goal is ill-defined in that the robot is provided with an over-generalized goal. In this situation, the Deliberator must expand the goal definition to include a set of subtasks. To consider an example, suppose the robot is required to retrieve an object. The robot is presented with "put a at (x, y)". It must decompose this goal to tasks that the Sequencer can handle. The Deliberator generates the series of subtasks "find a, lift a, go to (x, y), drop a". The Sequencer understands these tasks, and generates the behavior hierarchy for each, performing them in order to fulfill the goal. For this to work, the Deliberator must have the capability to develop an ordering for the task allocation, requiring a planning capability such as Partial Order Planning [47]. It also should generate new goals from sensor data processing that occur during the performance of a task, if necessary. If implemented in a single robot system, this task decomposition and planning structure provides all necessary requirements for a fully functional and sophisticated architecture. Deliberators on robots that are part of a Multi-Robot System (MRS), however, have additional requirements.

The description of HAMR to this point has not included discussion about other agents, as the lower levels of HAMR do not require additional information from other actors in the environment, with the exception of using sensor data for collision avoidance of the other robots, etc. HAMR at the lower levels is sufficiently generic and

shielded from outside influences, so the low-level architecture applies to both single robot and multi-robot systems. The task allocation responsibility of the Deliberator, however, indicates the need for knowledge of the actions of other group members. Thus, in order for each robot to contribute to the MRS, coordination is required so tasks and resources are appropriately allocated among the different group members. This is a challenging class of problems, and [18] and [34] each provide approaches to handle this. The nature of the implementation of HAMR discussed in the next chapter prevents any requirement for this level of solver. Additionally, the Deliberator must react to failure of tasks and adjust the assignment when alerted by the Sequencer. If, in the example presented above, the item is too heavy, the Deliberator determines that it needs coordination on the task and requests help from another robot. This is an additional requirement of the Deliberator, again only present if the robot needs to operate as part of an MRS. If acting independently, the task is simply recorded as a failure and there is no way to complete the task. Also recall that, as discussed in Chapter II, the system requirements include a fully decentralized architecture, so this processing is carried out internal to every member of the group. In other words, each robot must, upon receipt of the other robot's utilities, perform the processing to generate global task or resource assignments and individually determine their own allocation. This requires updating the state representation and generating the robot's utilities. These responsibilities are handled by the additional layer developed to enable improved multi-robot control, the Coordinator.

3.5 Coordinator Structure

Coordination processes are characterized by the two aspects of coordination: collaboration and cooperation. Collaboration in this sense is when multiple robots work together, each on a separate subtask without shared resources, to accomplish a task. There is little or no synchronization required between the robots. An example of this occurs with two robots cleaning a building. Each robot is assigned a static collection of rooms to clean and is unaffected by the other robot's progress.

Cooperation is when multiple robots work together to accomplish a task or subtask. These require some degree of synchronization and make use of shared resources, so the progress towards the final goal is mutually dependent.

These aspects of coordination are handled differently based upon the nature of the robots in the collective and the architectural approach applied. For example, ALLIANCE [42] makes use of internal behavior motivations, namely impatience and acquiescence along with communication. This provides coordination by modeling task progress. If a task is not proceeding appropriately, a robot becomes impatient and focuses on the task. Another robot, if attempting a task but failing, acquiesces and allows the first robot to take over. Cooperative activity is achieved by the same mechanism, where a robot's impatience adds it to the group working to complete the task. RoboSkeleton [13] uses a centralized CoachAgent to model adversarial activity and generate play strategies, which are then dictated to the active player agents in the system. This coordination is handled from a centralized aspect of the architecture. CAMPOUT [27] treats coordination as an extension of behaviors present in each agent. This assigns a leader and a follower, with the follower making use of shadow behaviors to basically copy the actions of the leader, thus providing tightly coupled cooperation. CAMPOUT assumes that loosely coupled cooperative and collaborative activities are effectively individual activities. The layered multi-robot architecture designed by Simmons [48] provides coordination through activation of behaviors in other robots. This allows a leader to control every aspect of the contributing robot's behaviors, thus enabling tightly coupled cooperation and collaboration. With the exception of ALLIANCE, these examples require a fairly extensive knowledge of the other robots in the group. ALLIANCE's approach enables moderately coupled cooperation without extensive knowledge, but also precludes much of the negotiation associated with task distribution. The approach applied in this document provides negotiation and also allows for mechanisms similar to those in ALLIANCE, where task progress dictates the need for more contributors. Therefore, in order to provide

this coordination yet still maximize independence, HAMR contains a layer called the Coordinator.

The Coordinator is a separate layer developed for enabling multi-robot control and contributing to a robust collection of mostly independent robots. It has three primary functions: first, it provides feedback to the Deliberator in order to aid the Deliberator in decision making for multi-robot task allocation by generating utility values for tasks. The second function is monitoring of the state representation to determine if the state has changed significantly enough to justify an update to the system's tasking. The third function is to maintain the state representation in order to incorporate accurate modeling of the other group members, important environmental data, and maintain global task and resource allocation records.

In order to perform the first function, the Coordinator possesses a means of determining a sequence for performing a task that has been decomposed by the Deliberator and the ability to assign a utility to that task. The Coordinator first takes a candidate task from the Deliberator, then determines, in much the same way as the Sequencer, the behaviors and skill sets the robot has available to perform this task. In contrast to the Sequencer, however, the Coordinator does not provide the set of tasks to the Controller in order to physically perform. Rather, it generates a utility value for this task based upon expected resource expense and general fitness. The utility calculation is shown by:

$$Utility(i) = \begin{cases} fitness(i) - cost(i) & \text{if} fitness(i) > cost(i); \\ 0 & \text{otherwise.} \end{cases}$$

Where i is the target task task, $fitness$ (or quality) is the expected value in having this robot perform the task, and *cost* is the expected resource expense of performing the task [24]. If the robot is unable to complete a task, $fitness$ is very small and causes a value of 0 for utility. This value is stored in the state representation as part of the task and the same procedure is followed for each candidate task. When complete,

the Coordinator transmits the overall goal utility to the other group members so their Deliberator layers can evaluate the goal and determine appropriate allocation of tasks and resources. This is performed using auction methods such as those presented in [23] [7]. The task is stored in the state as a goal, with an associated utility value and the agent assigned to the task. When the task is unassigned, a message is sent to the appropriate group members to generate an assignment. Storing the task in this manner reduces processing time if task reallocation is required and, if a task priority is incorporated, serves to create an internal task hierarchy, where tasks are ordered based upon priority and avoidance of clobbering.

During performance of the assigned task, the Coordinator monitors and maintains the state representation (the updated structure of HAMR is shown in Figure 3.4). This requires an expansion of the version of the state monitored by the Sequencer and maintained by the Controller, as this state representation also consists of the states of other robots, which is the third function discussed above. This information is stored according to each group member, with the state of the other robots represented by the information delivered from both sensor data and inter-robot communications. The information stored includes positions, task assignments, and resource allocations of the other group members. There is no need to store the capabilities of the other robots, since each robot determines its own capabilities independently. Whenever the state changes in an appreciable manner to the task at hand or to the global task allocation, the Coordinator alerts the Deliberator and a new allocation is determined. Changes to the state that are considered appreciable would include events like receiving notification that another robot is unable to complete a task, modifications to the state that eliminate the need to complete a task (such as finding an item in the course of exploring an area), or reaching an expiration time for task completion.

The communication requirements for HAMR are low. If extensive modeling of other agents is employed, the communication requirements are reduced further. In order to send the utility values to the other team members, a simple addressing scheme is used, requiring a single message generation that contains the task descrip-

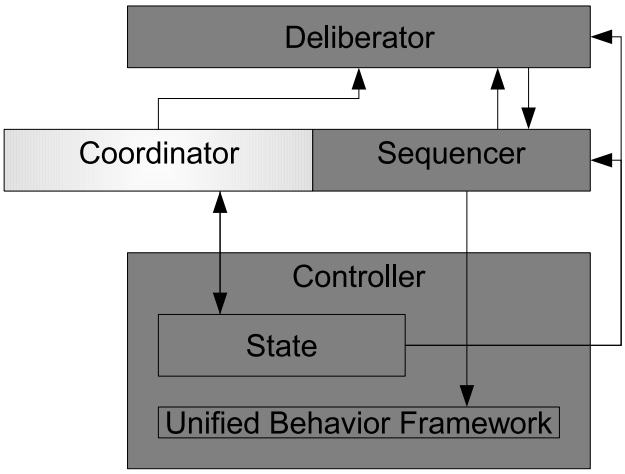

Figure 3.4: HAMR block diagram, including the Coordinator. The arrows indicate information flow direction.

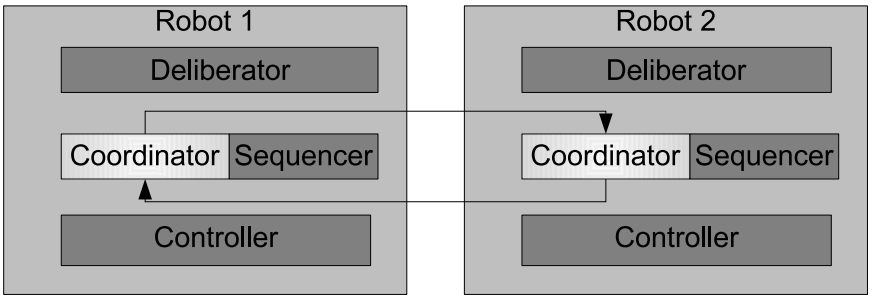

Figure 3.5: HAMR with multiple robots. The internal data flow has been removed for simplicity.

tion and utility values for the tasks. If this message is sent to only the other potential candidates for performing the task (subgroups within the collective), it reduces communication overhead and processing time for each team member. When a task is completed, a simple message consisting of the task description and a completion flag is sent to the appropriate team members. This provides a means for appropriately processing the tasks in any required order and in a timely manner, since the ordering was already determined by the Deliberator and the expiration time is determined with arrival of the task. HAMR is shown with inter-robot communication in Figure 3.5. The class structure of HAMR is shown in 3.6.

The Coordinator is crucial for MRSs using the TLA in that it provides a simple, standardized approach to enable coordination. It is present on all team members, so

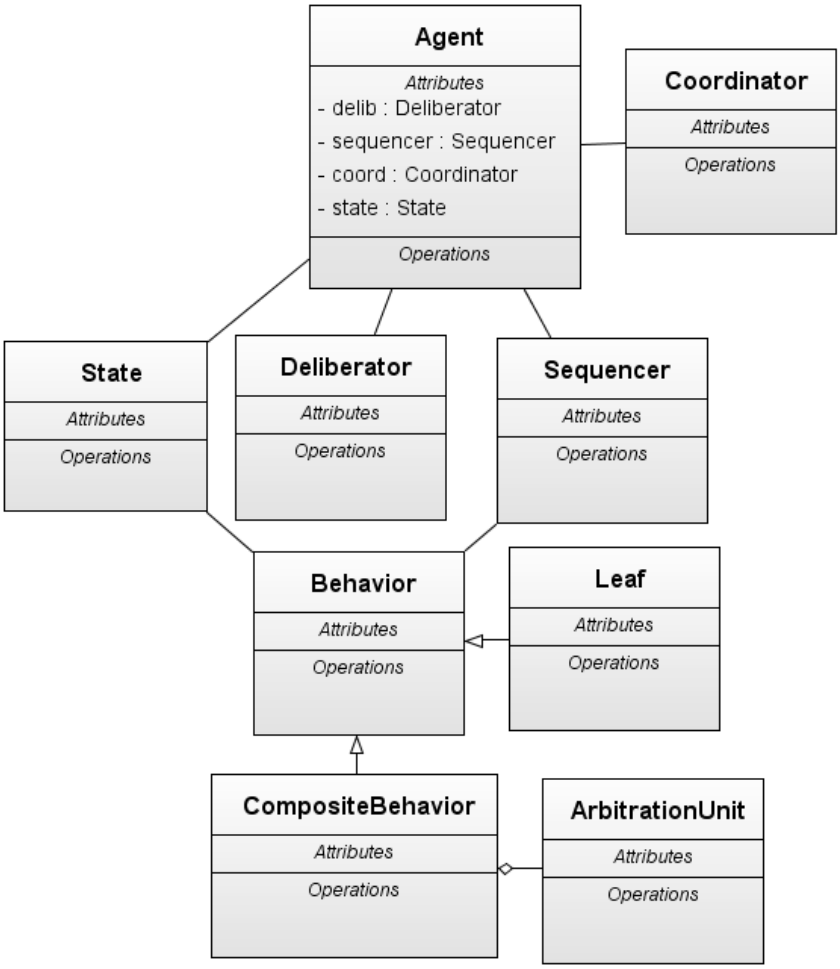

Figure 3.6: HAMR class structure. The Agent contains the Deliberator, Sequencer, and Coordinator layers, and the Sequencer generates the UBF Hierarchy.

it contributes to a fully decentralized system. It is usable in MRSs with heterogeneous robot teams, since the coordination is based upon utility values and, though it requires a small amount of platform dependence due to the need to determine fitness and resource expense, is still as independent as the UBF. This is enabled since the behaviors themselves may incorporate expense calculations and reduces the hardware dependence to a behavior-dependent function call, shielding the Coordinator from direct hardware-level access. It provides low bandwidth communication, since a single message is broadcast per robot describing utilities for all current tasks that those recipients that have the potential to also participate in the tasks intercept and process. During system design, a common message formatting scheme is determined, but this remains platform independent provided the communication techniques are the same from one robot to another. The Coordinator also contributes to a robust system, because each agent is individually capable and there is no cross-agent inhibition or activation that could cause erroneous behavior such as in the architecture described by Simmons, et al [48].

3.6 State

Though not a layer in the control system architecture, the State is an important and crucial component of HAMR. The State provides information to all layers, and the information provided limits and modifies the robot's behavior. The sensor data is stored here as raw or processed data (or a combination of the two), depending upon the sensor type. This includes odometry readings, range data, corrected position information, map data, etc. The State also stores goals in one of two categories: group goals and individual goals. The group goals are discussed above in Section 3.5, and the individual goals are often subcomponents of these group goals. They reflect currently active tasks and the subgoals that the individual robot must complete. This allows the Sequencer to generate goal-fulfilling behavior sets in order to complete the individual goal-related tasks. The State stores outbound communication data (created by the Coordinator) until the hardware sends it. Since the State is a means

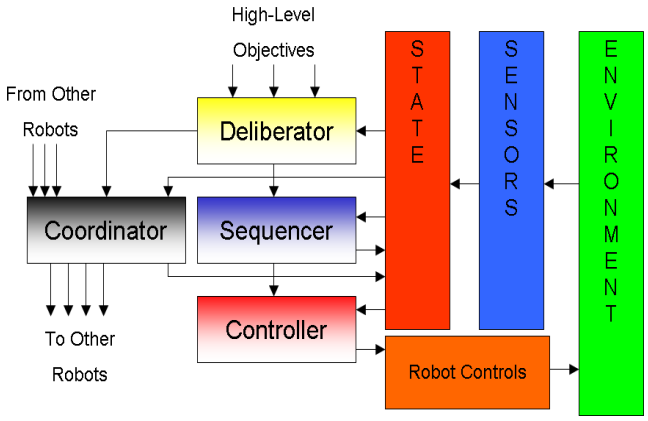

Figure 3.7: HAMR Data Flow. The task comes through the Deliberator and is decomposed into plans, behavior sets, then motor controls to affect the environment.

of cross-layer communication, it is important to ensure that the permissions for each subset of the state are appropriately assigned in order to avoid concurrency issues. To handle this, only sensors write sensor data to the State, but all layers have access to read the sensor data. Only the Coordinator adds communication messages, and only the communication hardware removes them when sent. In general, the specific responsibilities of HAMR layers limit the State access, and these limitations serve to ensure avoidance of concurrency issues. The full information flow regarding HAMR is shown in Figure 3.7.

3.7 Summary

HAMR is developed in order to fulfill the single agent and the two multiple agent requirements sets generated in Chapter II. By the expansion of the Single Robot Architecture (SRA) to include the Coordinator layer, HAMR is more robust than the architecture described by Simmons et al [48], since it reduces the potential of damaged or malfunctioning robots to adversely affect the entire collective. The contributions of

Table 3.1: Classification of HAMR via Cao's taxonomy.

	Centralization	Differentiation	Com Structs	Modeling
HAMR	decentralized: all robots are independent	heterogeneous: utility values platform independent	comm: can transfer info	strong: lower comm. overhead
Target	decentralized	heterogeneous	comm.	moderate

Table 3.2: Classification of HAMR via Dudek's taxonomy.

	Size	Range	Topography	Bandwidth	Reconfigurability	Proc Ability	Composition
Final Arch	LIM	NEAR	ADD	LOW	COM	TME	HET
Target	LIM	NEAR	ADD	LOW	COM	TME	HET

the Coordinator layer also includes greater ease of expansion for adding more robots to the collective, since the collective only needs to add the communication address of the new robot in order to fully incorporate the new robot into it. Furthermore, HAMR maintains modularity by keeping each robot from needing to know the full capabilities of the other robots in the collective, as the determination of utility values serves to provide a hardware invariant interface for determining those capabilities. The key advancements of this architecture also include provisions for advanced coordination capabilities through an emphasis on individual autonomy, contributions to this coordination by low communication requirements, and taskability for all associated robots in the collective. HAMR fulfills all MRS requirements generated based out of both Cao's taxonomy (RS1) [14], Table 3.1, and Dudek's taxonomy (RS2) [19], Table 3.2, and the single-agent requirements discussed in Section 2.2. In Table 3.1, the strong modeling is achieved by tasking: the other agents are modeled based upon their current tasking, allowing the agent to infer positions and task progress of the others. Furthermore, the considerations mentioned in Section 2.5 have been fulfilled by a combination of additional contributions from the Deliberator and the addition of the Coordinator layer. The following chapter discusses implementation of HAMR in a common MRS environment, the RoboCup Soccer Simulator.

IV. Simulation Environment

The RoboCup Soccer Simulator (RCSS) is a common multiagent testing platform designed to provide a real-time simulation of Multi-Robot Systems (MRSs). This chapter describes the RCSS and the implementation of the Hybrid Architecture for Multiple Robots (HAMR) within the RCSS. HAMR's description, provided in Chapter III indicates a general approach to the architecture, but does not describe an application of HAMR to a particular domain. This chapter, then, shows design variations and implementation specifics in order to generate an appropriate application of HAMR. In order to provide this information, the first section describes the RCSS in general, including its components and general makeup. The second section describes the architecture implementation specifics. The second section is divided into four subsections, each describing the implementation of a layer of HAMR.

4.1 RoboCup Soccer Simulator (RCSS)

The RCSS is a program designed to enable implementation and testing of multiagent systems on the task of playing soccer. The RCSS consists of four packages: base, server, monitor, and log player [15]. The base code package provides the common code used by the other packages. The server runs the simulation with clients sending commands and receiving sensory information. The monitor, shown in Figure 4.1, provides a view of the simulation. Each soccer game is played in two halves, each running 3000 cycles, or about five minutes. A sudden death round is played in the case of a tie at the end of the game, where the first goal scored wins the game.

The key part of the RCSS is the server. The server handles all environment updates, from both player actions and general environmental information. This includes ball movement after a kick and relocation of the ball on penalties or goals. The server also generates stamina, recovery, and effort values for each player. The players and coaches communicate with the server in the form of User Datagram Protocol (UDP) messages. Text-based commands are passed into the server with simple parameters and are processed. The server generates a "sense_body" message, which provides

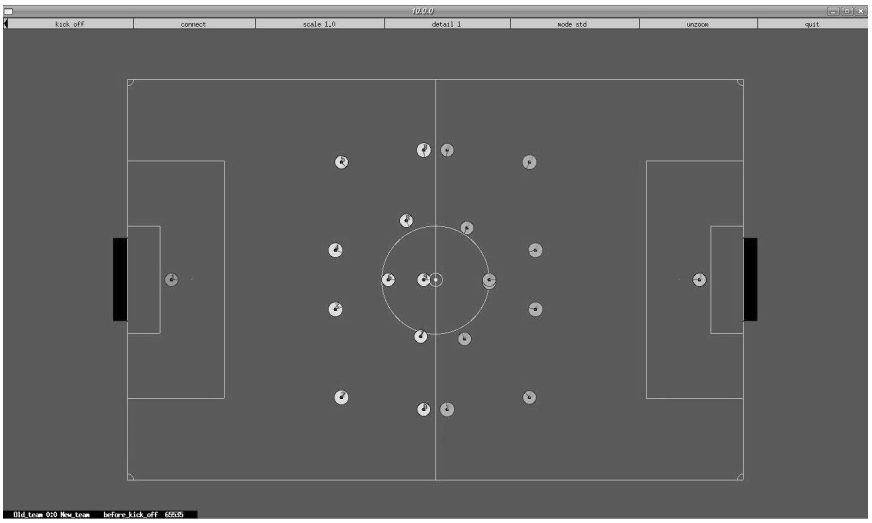

Figure 4.1: Screenshot of the RCSS with players distributed on the field in two teams.

the player and coach with sensory information. The server allows "Kick", "Dash", "Move", "Say", "Turn", "Catch", and "TurnNeck" commands from each player. The server handles these commands on a discrete time interval basis, processing one command per cycle per player. Since these commands are fairly generic and the server generates stamina, recovery, and effort values, the server allows for teams of heterogeneous player types. Each player type has different maximum values for stamina, recovery, and effort. In order to monitor the gameplay, the server contains a referee that enforces rules such as out of bounds and offsides.

Central to viewing a game in action is the monitor. This is a windowed application that shows the pitch (field), players, and the ball. The ball is displayed as an open white circle, with a filled inner white circle. Each player is shaded according to their team, and each player has an associated player number. The red line indicates the player's orientation, this determines what the player can sense. A player and the ball are shown in Figure 4.2

In order to develop and test a control architecture, developers create players and coaches. The players are the primary agents within the environment, performing actions on the field that influence ball movement and scoring. Depending upon the

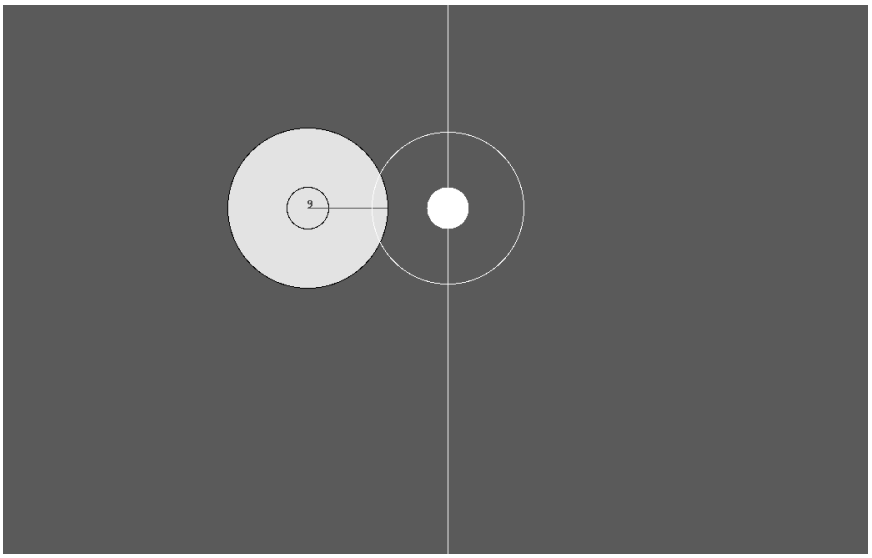

Figure 4.2: Partial screenshot of the RCSS showing a player and the ball.

decisions made for the degree of centralization of the resulting system, the role of coaches may range from practically nonexistent to full control. Since HAMR requires full decentralization, the implementation makes use of a functionless coach. The simulator itself does not provide the teams, so the teams must be developed from the ground up, passing messages to the server portion of the RCSS as described above.

4.2 HAMR Implementation

The team development extends the Trilearn Base Code from the Universiteit van Amsterdam's UvA Trilearn 2003 team [31]. This base code provides an implementation of the agent-environment synchronization, world model, and player skills, but decision making and coordination methods are not present [31]. The code establishes a good basis for handling much of the command generation, sensor feedback handling, and general underlying capabilities. HAMR is built from the bottom up, starting from the base code and building the Controller, Sequencer, Deliberator, and Coordinator layers.

4.2.1 Controller. Since a full system for the sensor feedback and motor command processing is present in the Trilearn Base code, the only portions of the Controller that are separately developed are the behaviors and their integration into the UBF. This section discusses their creation.

4.2.1.1 Behaviors. Procedural-based behaviors come with the Trilearn base code. These behaviors generate actions only when called by the player loop. Since the UBF operates on classes and makes use of polymorphism, the behaviors are changed to classes inheriting from the **Leaf** superclass, which is a subclass of the abstract class **Behavior** (see Section 3.2). There are many behaviors, each with its own specific applicability in the system. They are:

- **GoTo**: sends agent to the position indicated by the High-Level World Model (HLWM).
- **AlignNeckWithBody**: changes the agent's looking direction to be square with the body.
- **TurnBodyToPoint**: rotates the agent's body to the central point of the field.
- **TurnBackToPoint**: rotates the agent's body such that it is facing 180 ° from the central point of the field.
- **TurnNeckToPoint**: rotates the agent's neck to the central point of the field.
- **SearchBall**: rotates the agent's neck and body in order to locate the ball.
- **DashToPoint**: runs the agent to the central point of the field.
- **FreezeBall**: if the ball is within kickable range, it holds the ball still.
- **KickBallCloseToBody**: if the ball is within kickable range, it kicks the ball a short distance.
- **AccelerateBallToVelocity**: performs multiple kicks to make the ball move faster.
- **CatchBall**: a goalie-only behavior, catches the ball if it can be caught.

- **Communicate**: has the agent say the text phrase stored in the communication string state variable.
- **TeleportToPos**: moves the agent directly to its position according to the current formation type. Not available during gameplay.
- **ListenTo**: pays attention to another agent.
- **Tackle**: takes the ball away from an opposing agent.
- **TurnBodyToObject**: rotates the agent's body to face the ball.
- **TurnNeckToObject**: rotates the agent's neck to face the ball.
- **DirectTowards**: kicks the ball in the specified direction.
- **MoveToPos**: moves the agent to its position according to the current formation type.
- **CollideWithBall**: dash towards and run into the ball.
- **InterceptClose**: intercepts the ball if it can be intercepted within two cycles.
- **InterceptCloseGoalie**: intercepts the ball if it is close, goalie only.
- **KickTo**: kicks the ball to a specified target. The target can be a agent or the goal.
- **TurnWithBallTo**: rotates the agent's body while also keeping the ball to the agent's front.
- **MoveToPosAlongLine**: moves to a specified position along a line drawn between two objects.
- **Intercept**: intercepts the ball.
- **Dribble**: makes a series of kicks in order to move the ball and maintain possession.
- **DirectPass**: pass the ball directly to an object.
- **LeadingPass**: pass the ball to an object with some lead distance.

- **ThroughPass**: pass the ball through an area to an object.
- **OutplayOpponent**: attempt to kick the ball behind an opponent and recover it.
- **ClearBall**: kick the ball so it is removed from an area.
- **Mark**: defend against a agent, man-to-man style.
- **DefendGoalLine**: stands on the goal line to intercept any shots.
- **InterceptScoringAttempt**: intercept the ball when an opponent is trying to score.
- **HoldBall**: holds the ball. This is a goalie behavior only.

These behaviors all contain a *go* function, where the parameters fed to it are appropriate to the current state represented by the WorldModel (WM) and High Level World Model (HLWM) state files. The *go* function generates the action from the Trilearn Base Code's procedural behaviors, and varies widely from one behavior to another in both complexity and parameters passed to it. The behaviors also each have a specialized *genAction()* function, which is used by the UBF in order to generate commands. The *genAction* function examines the state, generates parameters to pass to the *go* function, and returns the response of the *go* function along with an associated vote for the perceived utility of implementing the behavior.

4.2.1.2 UBF. The UBF is incorporated as shown in Figure 4.3, with the notable exception that the behaviors are not passed a state for the genAction function. Instead, each behavior is passed a reference to the single WM and, if necessary, HLWM upon creation of an instance. This way, the *genAction()* function is parameterless and processes more quickly, since the entire state does not need to be placed on the stack. The arbiter is based simply upon utility, where the best behavior (according to the behavior's utility values) is selected. Unfortunately, due to the constraints placed upon the messages that can be passed to the Soccer Server, only one behavior is performed at a time, thereby limiting the arbiter to a winner-

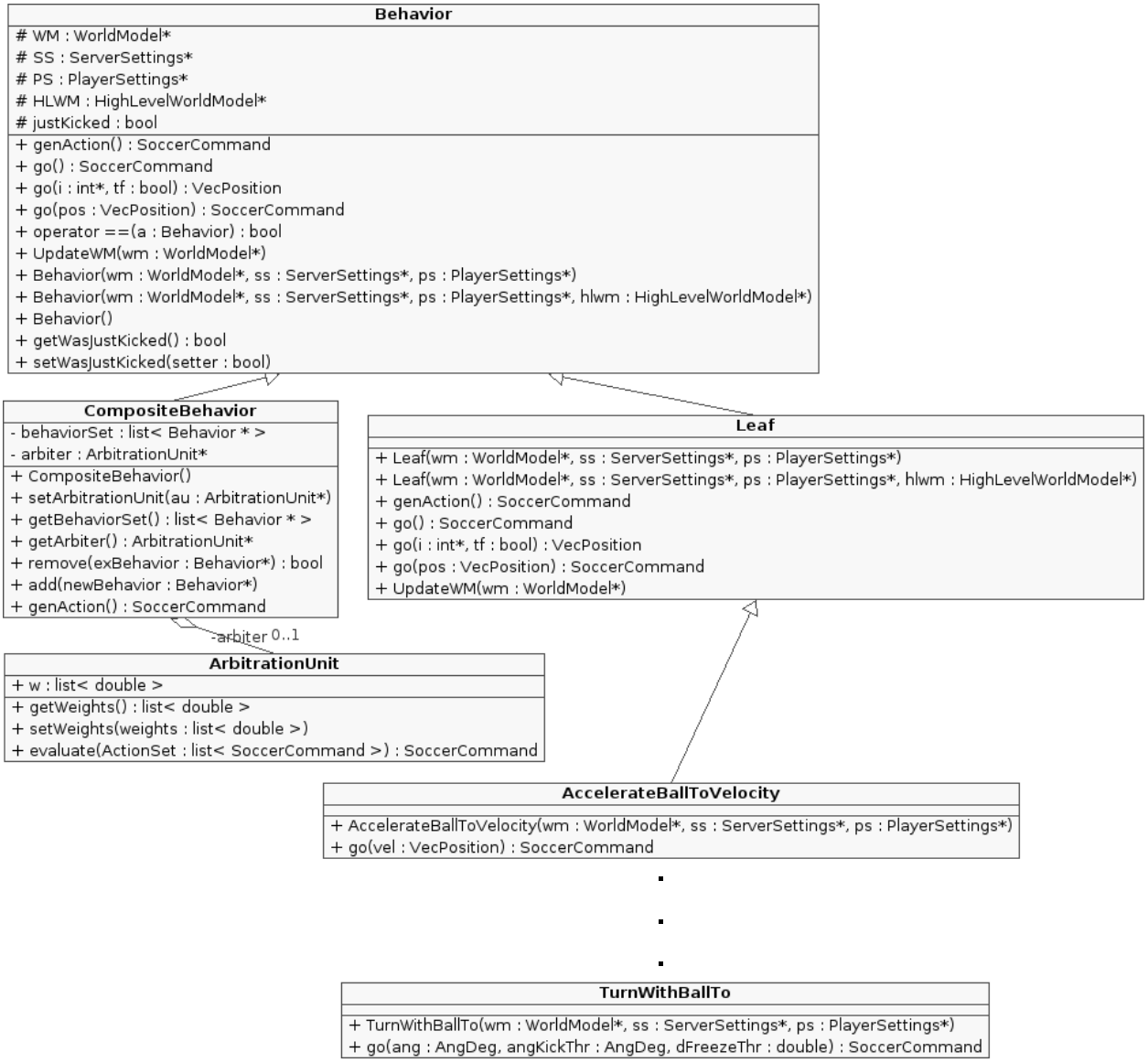

Figure 4.3: The internal structure of the UBF. Using an arbiter, a composite behavior is generated that makes use of multiple simple behaviors. [56]

take-all or command fusion scheme (provided the commands are in the same actuator group as defined by the RCSS). Therefore, though it is physically possible for a real-world soccer player to both kick the ball and turn his neck, these are issued as separate commands in the RCSS environment, so an RCSS player cannot. All of the behaviors listed in Section 4.2.1.1 fall under the **Leaf** superclass, which contains a generic *genAction()* function and constructor. The **Behavior** class, a superclass of **Leaf**, is also generic. The **CompositeBehavior** class contains the same variables and functions as in Figure 4.3. The behavior generated by the **CompositeBehavior** class is the one with the highest vote, since the arbiter performs a simple utility selection. A class diagram with the specific implementation of the UBF is also shown in Appendix A, Figure A.5.

4.2.2 Sequencer. The Sequencer acts as a container for all behaviors and also performs the selection of behaviors to activate through generation of the behavior hierarchy. Its primary activity is carried out in the runPlay function, which adds appropriate candidate behaviors to the **CompositeBehavior** instance based upon an examination of the WM and the current play. For example, if the ball is within range to be kicked by the agent, the Sequencer adds **KickTo**, **Dribble**, and **DirectPass** to the **CompositeBehavior** instance.

The Sequencer also contains an instance of the **Play** class, which is a container for the formation, current value for the play type, and preferred ball handler. The **Play** instance is generated by the Deliberator, and acts as a means of passing commands from the Deliberator to the Sequencer. The formation is the structure of soccer players on the pitch (field) and is based upon traditional soccer formations, those developed are the 4-2-4, 4-4-2, 4-3-3, 3-3-4, 2-4-4, and a kickoff formation which is similar to the 4-2-4. The play type enumeration allows the following values:

- RECLAIM_BALL: the opposing team has the ball, and the current play is designed to create a turn over. This normally requires a defensively-oriented formation such as the 4-4-2.

- TRAP: attempt to force an offside against the other team. If successful, this play type results in a turn over.
- DRIVE: the agent's team has possession of the ball and is moving it down the field. This normally works best with a 4-3-3 or 3-3-4 offensive formation.
- SCORING: the agent's team has possession of the ball and is attempting to score. This normally works best with a 2-4-4 formation.
- KICKOFF_US: kickoff for the agent's team. Works best in a kickoff formation.
- KICKOFF_THEM: kickoff for the opposing team.
- CORNER_KICK_US: corner kick for the agent's team. Normally works best with a 3-3-4 offensive formation.
- CORNER_KICK_THEM: corner kick for the opposing team. Normally works best with a corner kick formation.
- PENALTY_US: the agent's team has a corner kick.
- PENALTY_THEM: the opponent's team has a penalty kick.
- GOAL_KICK_US: the agent's team has a goal kick.
- GOAL_KICK_THEM: the opposing team has a goal kick.
- FREEKICK_US: the agent's team has a free kick.
- FREEKICK_THEM: the opposing team has a free kick.

These play types provide a readily accessible evaluation of the state for the Sequencer, such that the Sequencer can then spend more time activating the appropriate behaviors for those states. Each call of the *runPlay()* function compares the current play type with the **HLWM** play type, and if they don't match, it sets a flag that notifies the Deliberator that it is needed in order to reevaluate the state. This comparison is run once during each gameplay cycle. A class diagram of the Sequencer is shown in Appendix A, Figure A.4.

4.2.3 Deliberator. The Deliberator's only function in the RoboCup domain is the setting of the play type, formation, and preferred ball handler for the Sequencer. The high-level decision making is simple in soccer, since there is only ever two significant objectives: score a goal, and keep the opposing team from scoring. Subtasks can be decomposed from this, but it is unnecessary since the Sequencer can generate a simple plan which, due to its simplicity, is more robust to task failure. Because of this, the behavior activation decision making is left to the Sequencer. The Deliberator, instead, generates instances of the **Play** class. This is done via the primary function of the Deliberator, which is the *genPlay()* function. The *genPlay()* function evaluates the WM and HLWM, generates a **Play** instance, then sets that play to active in the Sequencer. In order to make use of the Deliberator as rarely as possible, the Deliberator's *genPlay()* function is only called if the Sequencer detects that the current play is done. The detection is performed by examining the **Play** instance, as the **Play** instance itself determines whether it is complete by determining current ball possession and position relative to tolerable values for the play. Additionally, the Deliberator contains the Sequencer instance that it interfaces with. This allows the commands to cascade downwards: the Deliberator generates the play, the Sequencer implements the play by activating component behaviors, and the behaviors physically carry out the activity. This is a minor variation on the hierarchical structure presented in Section 2.2, with the only significant change given by the functionality of the Deliberator.

4.2.4 Coordinator. The Coordinator relies heavily upon the HLWM, since it must generate and process appropriate communication messages and, in some cases, set a state variable based upon the received information. There are two important functions in this class: the *GenMsg()* and the *GenUtility()* functions. The *GenUtility()* function computes individual task utilities based upon received messages. One assignment it computes is which opponent to mark when playing defense. First, it generates a list of possible candidates and goes through the list,

removing any opponent already marked. If there are any remaining unmarked opponents, the *GenUtility()* function selects it, generates a utility value, and indicates both the utility value and the potential assignment in the WM. If all opponents from the list are covered, the *GenUtility()* function selects the candidate from the original list with the highest utility and covers it (double-teaming the selected opponent), also indicating it in the WM. This enables the defenders to coordinate their coverage assignments and more efficiently cover opponents and improve the chances for forcing a turnover. Appropriate allocation of these marking assignments requires communication, since the system should avoid double assignments if there are uncovered opponents. This communication is handled by the *GenMsg()* function. The utility value is calculated based upon distance to the target and current stamina levels generating the cost function and the player type indicating fitness. So $cost(target) = (DistanceTo(target) + StaminaUsed)$ and the utility value is:

$$Utility(target) = \begin{cases} (playerType * 10) - cost(target) & \text{if } (playerType * 10) > cost(target); \\ 0 & \text{otherwise.} \end{cases}$$

The physical communication is part of the Controller layer functionality, so the task of the *GenMsg()* function is to generate an appropriately formatted message in an efficient manner. The message is currently sent in one of three formats: WorldModel update, coverage assignment, and utility value formats. The WorldModel update format has two message structures: opponent attacker and ball status messages. The message structuring for all three formats is shown below.

- Opponent Attacker Message: The opponent attacker message transmits a variation on a time stamp along with the name and position of the deepest attacker and the offside line. The offside line runs parallel to the goal line immediately behind the nearest non-goalie teammate to the goal. The first byte is encoded by taking the current cycle (time point in the game) modulo 10, and adding that value to the ASCII character 'a'. Therefore, the first byte is between the lower

case letters 'a' and 'j'. The second byte is the offside line, which is encoded by a single ASCII character between 'a' and 'Z'. This is decoded as a numerical value between 0 and 52. The next character is the opponent player number added to ASCII character 'a'. The next three bytes are the x-coordinate, shifted left by one digit and displayed with no values to the right of the decimal. The y-coordinate is last, displayed similarly to the x-coordinate but shown in two bytes, unless negative. As an example, encoding opponent number 9 at position (-40.3345, -3.3123) and the offside line at 27.0 during cycle 1534 would give: "eBj403-33", where 'e' is 1534 modulo 10 + 'a', 'B' is 27, 'j' is 9 + 'a', and the coordinates are shifted left by one and the decimal values dropped.

- Ball Status Message: The ball status message, which begins with the same two characters as the opponent attacker message, uses eight bytes to encode the ball status. The first two of these are the x-coordinate of the ball and the second two are the y-coordinate of the ball. The next four bytes are the x and y component velocities of the ball, each using two digits.
- Coverage Assignment Message: The last message type is in the coverage assignment format. This message encodes the cycle in a similar manner as shown above. The remaining bytes are pertaining to the coverage assignment. The first one or two bytes of the remaining message is the player number, the next four bytes are the letters of the word "Mark", and the last one or two bytes is the opponent player number being covered. This is generated after communication of the utility values.
- Utility values: The utility values are transferred using the cycle as shown above, followed by the player number, the word "Ut" and the potential target for assignment. The final two digits are the utility value.

Upon reception of a communication message, the Coordinator makes the appropriate updates to the WM regarding the ball, opponent, coverage assignment, and other agent's utilities. Before making any of these updates, the value of the informa-

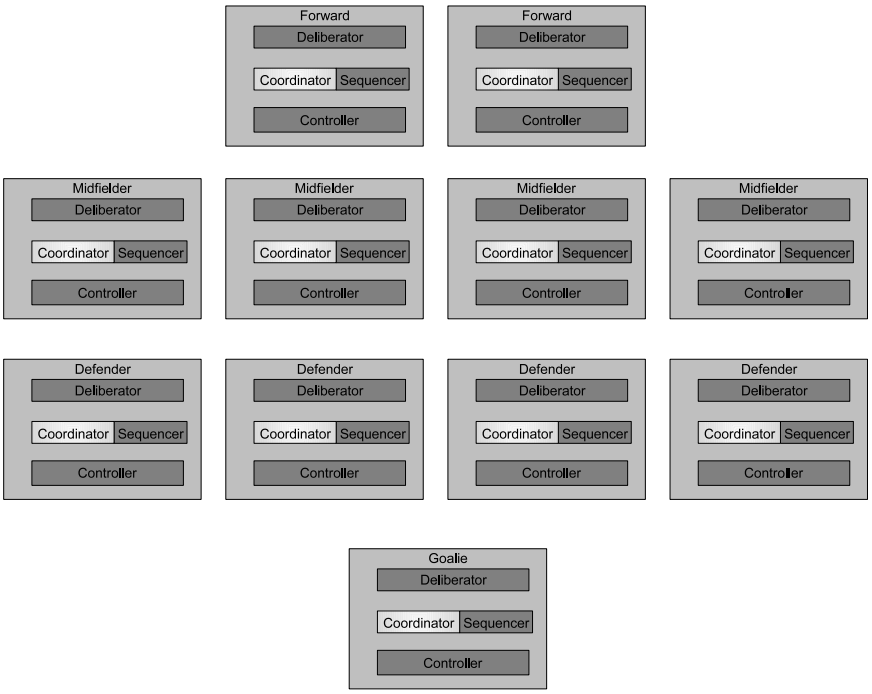

Figure 4.4: The team with each agent showing the architecture. The Coordinator layers communicate with each other.

tion is checked against the current WM. The update is kept if the data is newer than the last time local sensing received data, and replaced readily if local sensing receives additional data. Figure 4.4 shows the team configuration in a 4-4-2 formation with HAMR internal to each team member.

4.3 Summary

This chapter introduced the RoboCup Soccer Simulator and presented an approach to integrating HAMR as a team in the simulator. HAMR provides a highly modular, stable framework for behavior development and coordination on tasks. HAMR integrates well, with the UBF providing much of the structure for the Controller layer and the Sequencer generating the behavior hierarchy for the UBF. The **Play** class provides a means of information transfer between the Deliberator and the Sequencer while enabling an additional abstraction of the state. The Coordinator solves for individual allocation of tasks requiring coordination, and efficiently generates messages for the communication mechanisms contained in the Controller. Thus,

individual autonomy is preserved, communication bandwidth is kept low, and high-level coordination is provided. In order to show the performance of HAMR in the RCSS domain, Chapter V presents comparison of the results of integrating HAMR versus the Layered Multirobot Architecture presented in Section 2.3 and also testing HAMR against the Trilearn Base code.

V. Results

Testing of a design approach is difficult, since few benchmarks exist to validate the approaches. The most effective way to test the performance of the Hybrid Architecture for Multiple Robots (HAMR) is to implement an alternate architecture and comparatively test the two. The alternate architecture tested against HAMR is Simmons' Layered Multirobot Architecture [48]. As discussed previously, the Layered Multirobot Architecture fulfills most of the architecture requirements. It makes use of the Three Layer Architectural approach (the Single Robot Architecture (SRA) requirement). It is fully decentralized, allows for heterogeneity, communicates through both sensing and communication, and has strong modeling of other agents, fulfilling Requirements Set 1 (RS1). It contains a SIZE-LIM collective size, uses COM-NEAR communication range, makes use of a TOP-ADD communication topology, has ARR-COM configurability, assumes a PROC-TME processing equivalent, and allows for CMP-HET composition, thus fulfilling much of Requirements Set 2 (RS2). In addition, centrally controlled and managed communication in HAMR is not present in LMA, which is estimated as the key factor in performance variations between the two architectures. The lone exception to this is that the Layered Multirobot Architecture uses BAND-HIGH communication bandwidth. Nevertheless, the Layered Multirobot Architecture is the closest multiagent architecture of those reviewed to fulfilling the requirements, thus making it a good candidate for the comparison.

The primary benefits of HAMR over the Layered Multirobot Architecture are its lower communication overhead and the lack of cross-activation of behavior sets, which is one of the structural design approaches used in the Layered Multirobot Architecture. Thus, one expects the comparison of the two to result in HAMR out scoring the Layered Multirobot Architecture, ceteris paribus (including behavior definitions and general hierarchical structure).

In testing, the implementations were run against each other for seven sets of thirty games, with score differentials recorded. Each set of games has different communication settings enabled on the Layered Multirobot Architecture. The results

indicate better performance on behalf of HAMR. This Chapter discusses these tests and describes using them to test the performance of HAMR versus the Layered Multirobot Architecture. In addition, the Trilearn Base code is tested against HAMR in order to determine any gains or losses to performance acquired by the hierarchical structure of HAMR. This test was also for thirty games.

5.1 Implementation of the Layered Multirobot Architecture

Of the existing architectures reviewed in Chapter II, the Layered Multirobot Architecture is the closest to fulfilling the desired requirements described by the single robot architecture requirements, Requirements Set 1, and Requirements Set 2. Thus, in order to evaluate the capabilities and contributions of HAMR, the Layered Multirobot Architecture is implemented in the RoboCup Soccer Simulator (RCSS) and pitted against the implementation from Chapter IV. This provides a means of measuring the performance of HAMR by evaluating communication overhead and relative score. The implementation of the Layered Multirobot Architecture has the same behaviors in the same manner, and the ball and offsides information messages are generated in the same manner as in HAMR, but at the Controller level. For the Sequencer, communication messages are generated that indicate the agent's active behavior set. The agent on the receiving end of the message processes this message and activates a complementary behavior set. The Deliberator generates messages indicating the play type, and the receiving agent also activates this play in order to have a more consistent play activation type. These communication methods are also shown in Table 5.1.

With any configuration (regardless of architecture), the communication is somewhat limited in that only one communication message can be sent per cycle. Thus, by necessity, messages in the Layered Multirobot Architecture are prioritized and supersede other messages if multiple messages are generated in one cycle. This prioritization is based upon the layers, with Deliberator messages more important than Sequencer messages and Sequencer messages more important than Controller mes-

Table 5.1: Communication methods in the Layered Multirobot Architecture implementation.

Layer	Message Type	When Produced
Deliberator	Play	At Play Generation
Sequencer	Behavior Set	When Behaviors Change
Controller	Ball Position	Every Cycle
	Deepest opponent position	Every Cycle

sages. In order to approximate a normal distribution, thirty games are run against HAMR with this configuration. Next, thirty games are run with only the Deliberator layers communicating, thirty with only the Sequencer communication, and thirty with only the Controller level communication. Finally, three sets of thirty games are run with the first thirty having both the Controller and Sequencer communication active, the next thirty with Controller and Deliberator communication, and the last thirty with Sequencer and Deliberator communication active. This way, a test of the full spectrum of all possible combinations of communication is performed. The results of these runs are discussed in the next section. In addition, thirty games are played in order to test HAMR against the Trilearn Base code. This is performed in order to determine the performance of the hierarchical nature of HAMR against an effectively implemented reactive architecture.

5.2 Results of gameplay

This section discusses the results of the games played between HAMR and the Layered Multirobot Architecture as described in Section 5.1, along with the games played between HAMR and the Trilearn Base code. The relative performance of the Layered Multirobot Architecture and HAMR are to be measured. To perform this measurement, thirty games are played with both architectures as presented in the associated literature (referred to as the base test). Blocked communication messages are also measured for these games to illustrate the extent of the constraints on the Layered Multirobot Architecture's communication structure. To examine the poten-

tial source or sources of any score discrepancy, communication capabilities are varied on the Layered Multirobot Architecture for a secondary block of tests (referred to as the secondary tests). Thirty games are played with communication only on the Deliberator layer, thirty with communication only on the Sequencer layer, and thirty with communication only on the Controller layer. The third test is performed with HAMR playing the Trilearn Base code for thirty games. All results are presented as a score differential in favor of HAMR (e.g. a value of 11 represents HAMR winning by 11 goals, -3 represents HAMR losing by three goals).

In order to determine the statistical significance of the score values, the z-test is used. The z-test assumes a normal distribution, so a test for normality is first performed using the χ^2 goodness-of-fit test [50]. The χ^2 value is calculated using the following:

$$\chi^2 = \sum_{i=1}^{N} (O_i - E_i)^2 / E_i \tag{5.1}$$

Where O_i are the observed counts in an associated bin and E_i are the expected counts in that associated bin. This tests the null hypothesis that the data in the associated vector are a random sample from a normal distribution. The null hypothesis is rejected if the χ^2 value is greater than 5 (based upon a 95% significance value). Table 5.4 shows the associated χ^2 and null hypothesis rejection conclusion. For better matching on the Controller, two outliers are removed. The normal probability density functions (PDFs) for these results are then compared using a z-test [43] in order to determine statistical significance and thus determine aspects of the contribution to the score differential in the first test set. The z-test is calculated using Standard Error (SE) [43], which is expressed as $\frac{\sigma}{\sqrt{n}}$, where σ is the standard deviation and n is the size of the sample under consideration. Next, the z score (z) is calculated using $z = \frac{x-\mu}{SE}$ [43], where x is the sample mean and μ is the population mean. Thus, the z score is

$$\frac{x - \mu}{\frac{\sigma}{\sqrt{n}}} \tag{5.2}$$

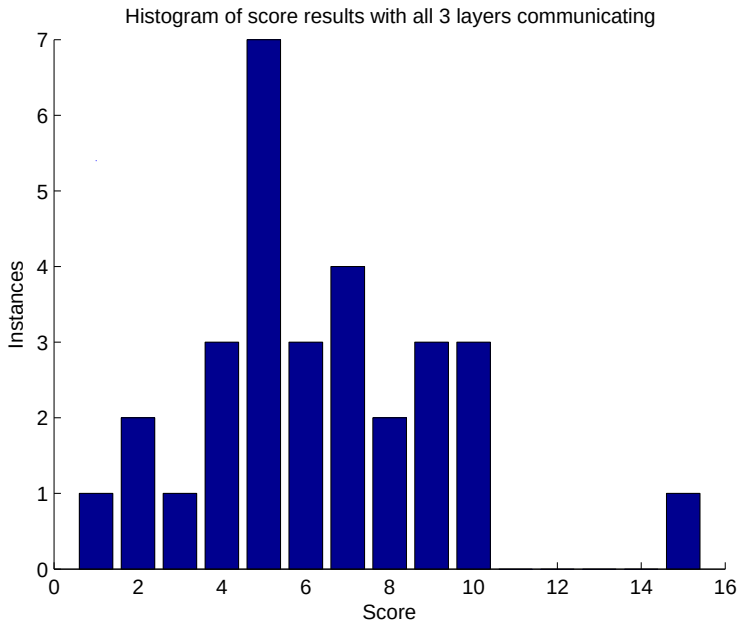

Figure 5.1: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on all layers, played against HAMR. LMA scored zero goals in every game.

or better expressed as

$$\frac{\sqrt{n}(x-\mu)}{\sigma} \tag{5.3}$$

Using this equation, the z scores are calculated.

5.2.1 Base Test. The base test (Figure 5.1) shows score occurrences relative to the score value with the Layered Multirobot Architecture against HAMR. This indicates a mean of 8.3 goals, with a standard deviation of 2.94, with HAMR winning every game. The blocked messages from these games consist of a mean of 58662.7 and a standard deviation of 4707.84. These blocked messages are the net sum of blocked messages over all players from the Layered Multirobot Archiecture. In all these games, the Layered Multirobot Architecture scored zero goals.

5.2.2 Secondary Tests. In order to better understand the source of the significant differences in score between the architectures, communication on the LMA implementation is reconfigured such that it is limited to only one layer communicating

at a time and thirty games are played at each configuration. The histogram in Figure 5.2 shows score occurrences relative to the score value with just the Deliberator layers communicating. This reflects a mean of 8.1 goals and a standard deviation of 2.5. Figure 5.3 shows the same for communication at just the Sequencer layer, reflecting a mean of 11.07 and a standard deviation of 2.5. This is the most significant of the individual layer communication tests. Figure 5.4 shows the histogram for communication at only the Controller layer for thirty games, indicating a mean of 3.37 and a standard deviation of 1.35. The two layer communication tests reflect similar results. The score occurrences with the Layered Multirobot Architecture communicating on the Controller and Sequencer layers are shown in the histogram of Figure 5.5. This is similar to the results with only the Sequencer communicating, with a mean of 11.1 and a standard deviation of 1.97. Next, the histogram in Figure 5.6 shows the results with communication on the Controller and Deliberator layers, with a mean value of 2.3 and a standard deviation of 1.12. Finally, Figure 5.7 shows the results with communication on the Sequencer and Deliberator layers. This indicates a mean of 11.4 and standard deviation of 1.28, also similar to the results of the Sequencer communication alone. The means and standard deviations of scores for each configuration as mentioned above are shown in Table 5.2. The most significant of the single layer and two layer communication score differences are shown in bold, which are the Sequencer for the single layer and the Sequencer and Deliberator communication for the two layer communication. This is due to the interference caused by the Sequencer layer communications in the Layered Multirobot Architecture.

5.2.3 Third Test. The tertiary test is performed between HAMR and the Trilearn Base code. The Trilearn Base code is implemented with little modification from the original source. The **KickTo** action generation is modified to always kick the ball at maximum kick power directly towards the opponent's goal, thus matching the general behavior structuring of the unmodified Trilearn Base code. The histogram of thirty games played with this configuration is shown in Figure 5.8. This reflects a mean

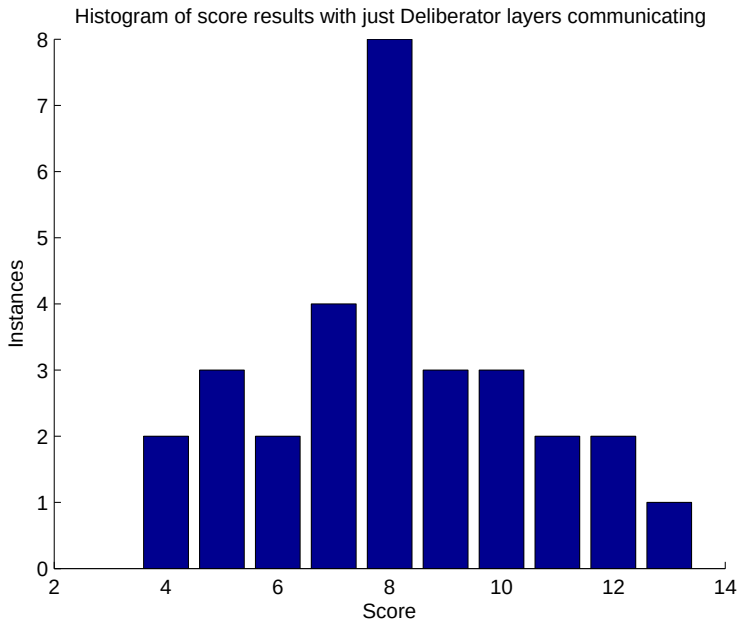

Figure 5.2: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on only the Deliberator layer, played against HAMR.

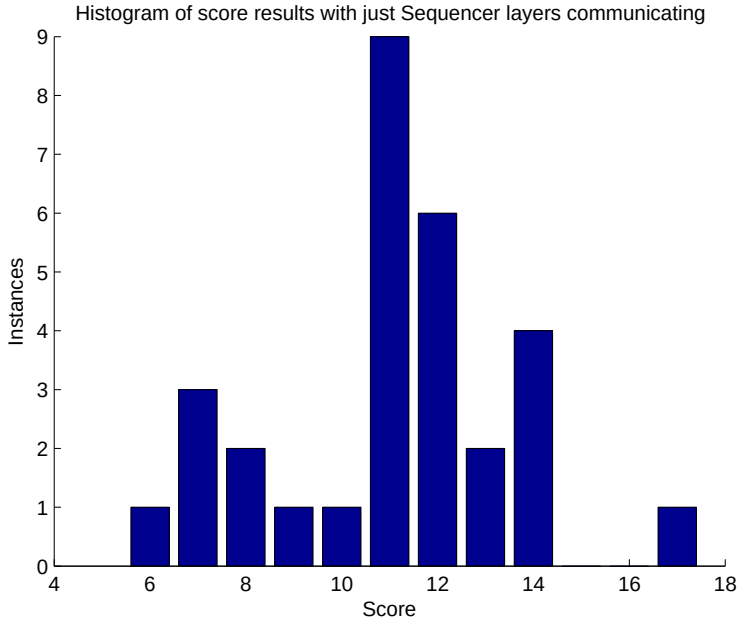

Figure 5.3: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on only the Sequencer layer, played against HAMR.

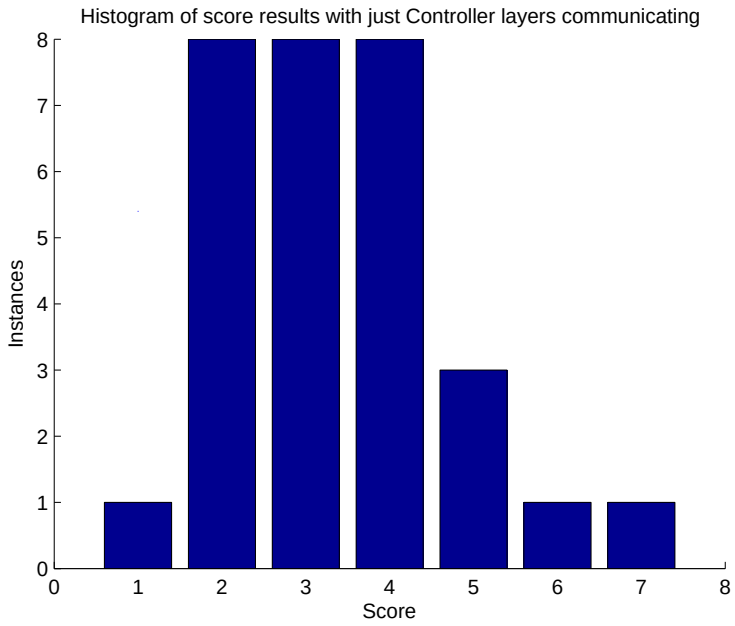

Figure 5.4: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on only the Controller layer, played against HAMR.

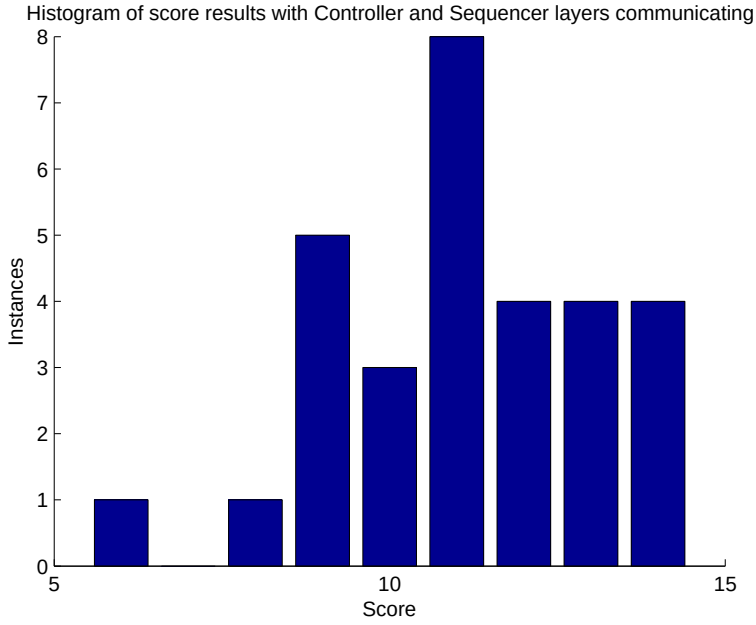

Figure 5.5: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on the Controller and Sequencer layers, played against HAMR.

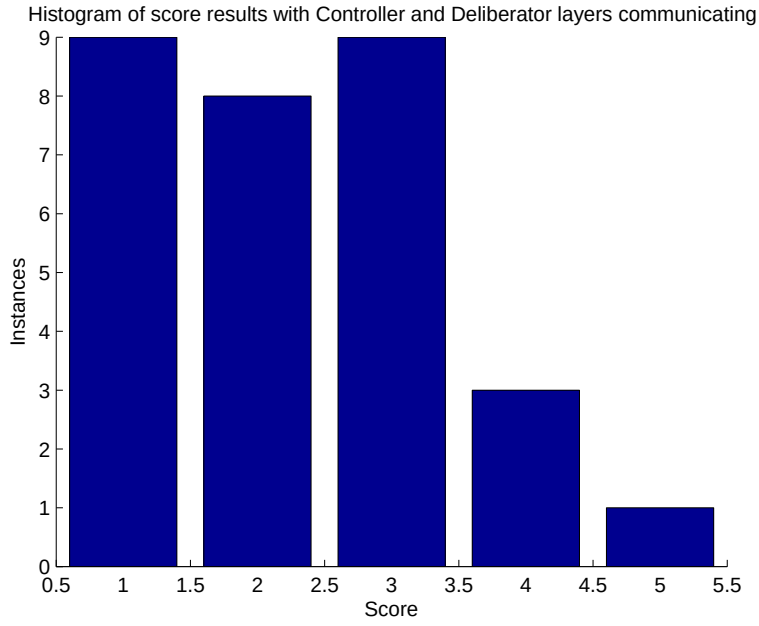

Figure 5.6: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on the Controller and Deliberator layers, played against HAMR.

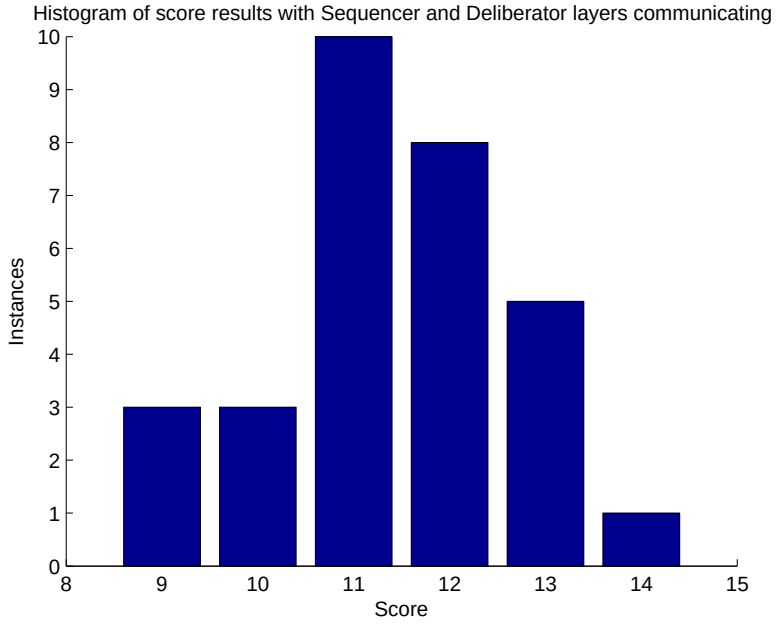

Figure 5.7: Histogram of thirty game score results with the Layered Multirobot Architecture communicating on the Sequencer and Deliberator layers, played against HAMR.

Table 5.2: Means and Standard Deviations of score differentials with regards to HAMR in the games played against the Layered Multirobot Architecture.

Communication	Mean Score	Standard Deviation
All Layers	8.3	2.94
Deliberator	8.1	2.35
Sequencer	**11.07**	**2.5**
Controller	3.37	1.35
Cont & Seq	11.1	1.97
Cont & Del	2.3	1.12
Seq & Del	**11.4**	**1.28**

Table 5.3: Means and Standard Deviations of messages blocked by the Layered Multirobot Architecture in the three-layer communication games played vs. HAMR.

Type	Mean	Standard Deviation
Blocked Messages-All Layers	58662.7	4707.84

Table 5.4: χ^2 goodness-of-fit results for HAMR vs. Layered Multirobot Architecture.

Communication	χ^2	reject null hypothesis?
All Layers	2.6096	no
Deliberator	0.5303	no
Sequencer	4.0096	no
Controller	1.4779	no
Cont & Seq	1.3732	no
Cont & Del	1.2617	no
Seq & Del	5.2854×10^{-4}	no

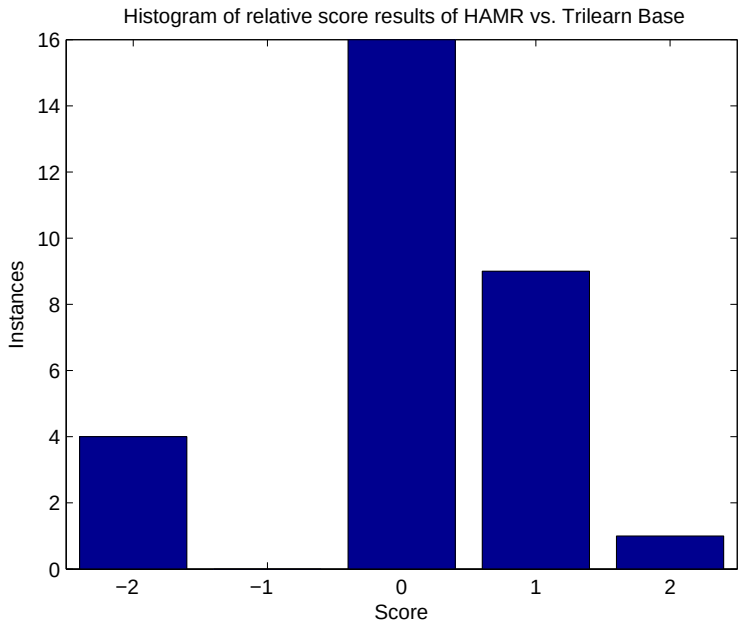

Figure 5.8: Histogram of thirty game score differentials with the Trilearn Base code against HAMR.

Table 5.5: χ^2 goodness-of-fit results for HAMR vs. Trilearn Base code.

Result Type	χ^2	reject null hypothesis?
Thirty games	5.1601	no

score of 0.10 and standard deviation of 0.9948. The χ^2 calculation is shown in Table 5.5. Its value is 5.1601. This indicates that the thirty games follow an approximately normal distribution and both HAMR and the Trilearn Base code perform roughly equivalently.

5.2.4 Analysis of Results. From the scores obtained from the base and secondary tests, the χ^2 goodness-of-fit test from Equation 5.1 is performed in order to test for normality. All results are approximately normal, and are shown in Table 5.4. Communication on all layers gives a χ^2 value of 2.6096. Communication on the Deliberator layer gives χ^2 equal to 0.5303. The Controller layer communication gives a value of 1.4779 for χ^2. For the two-layer communications, the Controller and

Sequencer together give a χ^2 value of 1.3732, the Controller and Deliberator give a χ^2 of 1.2617, and the Sequencer and Deliberator communication provide a very low χ^2 of 5.2854×10^{-4}. With these results indicating normal distribution is acceptable for all tests, the z-test is run on each combination of these two tests. The results of the z-test are calculated from Equation 5.3. The percentage of the population that falls closer to the mean is also determined from this, and if this value is less than 95%, it is considered to be part of the same distribution. If greater than 95%, the null hypothesis is rejected and it is considered statistically significant. These values are shown in Table 5.6. From this table, it is apparent that the Sequencer and Controller make unique contributions to the score result of all layers. This table also indicates that in most cases where Sequencer communication is enabled, it is not considered statistically significant relative to other instances where Sequencer communication is enabled. The z-test shows that the results are statistically significant except with regards to all layers communicating vs. the Deliberator only communicating, and in all instances other than all layers where the Sequencer is communicating in both contributors to the z-test. These are in Sequencer vs. Controller & Sequencer, Sequencer vs. Sequencer & Deliberator, and Controller & Sequencer vs. Sequencer & Deliberator. The z-score values are 0.2673, 0.2673, 0.2967 in these cases, respectively. The z-score for all layers communicating vs. the Deliberator is 0.1443.

5.2.5 Discussion. The primary cause of the score differential in the base and secondary tests is communication on the Sequencer layer. Note from Table 5.6 that in most cases where Sequencer communication is enabled, it is not considered statistically significant relative to other instances where Sequencer communication is enabled. In other words, the Sequencer communication dominates the effect on the Layered Multirobot Architecture's performance by causing erratic behavior and reducing its effectiveness in the RCSS domain. With this communication enabled, each agent receives messages pertaining to the behavior set of another agent, thus causing the agent to enable a complementary behavior set. However, with the highly

Table 5.6: Statistical significance calculations of score results (Layered Multirobot Architecture vs. HAMR), shown for each configuration of LMA.

Comparison	x	μ	σ	z	z-test	% closer	Significant
All Layers vs. Delib	8.1	8.3	2.94	-0.37	0.1443	28.86	no
All Layers vs. Seq	11.07	8.3	2.94	5.16	0.5	100	yes
All Layers vs. Cont	3.37	8.3	2.94	-9.18	0.5	100	yes
All Layers vs. Cont & Seq	11.1	8.3	2.94	5.22	0.5	100	yes
All Layers vs. Cont & Del	2.3	8.3	2.94	-11.18	0.5	100	yes
All Layers vs. Seq & Del	11.4	8.3	2.94	5.78	0.5	100	yes
Deliberator vs. Seq	11.07	8.1	2.35	6.92	0.5	100	yes
Deliberator vs. Cont	3.37	8.1	2.35	-9.37	0.5	100	yes
Deliberator vs. Cont & Seq	11.1	8.1	2.35	6.98	0.5	100	yes
Deliberator vs. Cont & Del	2.3	8.1	2.35	-13.50	0.5	100	yes
Deliberator vs. Seq & Del	11.4	8.1	2.35	7.67	0.5	100	yes
Sequencer vs. Cont	3.37	11.07	2.5	-16.87	0.5	100	yes
Sequencer vs. Cont & Seq	11.1	11.07	2.5	.0729	0.2673	53.46	no
Sequencer vs. Cont & Del	2.3	11.07	2.5	-19.17	0.5	100	yes
Sequencer vs. Seq & Del	11.4	11.07	2.5	.7291	0.2673	53.46	no
Controller vs. Cont & Seq	11.1	3.37	1.35	31.34	0.5	100	yes
Controller vs. Cont & Del	2.3	3.34	1.35	-4.32	0.5	100	yes
Controller vs. Seq & Del	11.4	3.34	1.35	32.55	0.5	100	yes
Cont & Seq vs. Cont & Del	2.3	11.1	1.97	-24.45	0.5	100	yes
Cont & Seq vs. Seq & Del	11.4	11.1	1.97	0.8335	0.2967	0.5934	no
Cont & Del vs. Seq & Del	11.4	2.3	1.12	44.55	0.5	100	yes

dynamic nature of soccer and its implementation in the RoboCup Soccer Simulator (RCSS), this causes a very rapid shift in behavior sets. This, on occasion, causes the agents to behave erratically, rapidly shifting between behavior sets and accomplishing very little in terms of ball retrieval or advancement of the ball towards the opponent's goal. Thus, this behavior set activation acts as an inhibitor, just as discussed earlier.

The Deliberator level communication causes less interference than that of the Sequencer, but it also fails to deliver a significant benefit. Since the messages passed at this layer are plays, it serves merely to define a preferred ball handler and a formation position for the agent. This does help to a certain extent, since all agents on the team have a universal play selected, but is otherwise limited in its contributions due to the limited amount of information contained within the play.

The Controller level communication contains many of the same aspects of the ball status and opponent player information as the Coordinator layer-generated messages, and thus serves to contribute more benefit to the Layered Multirobot Architecture implementation than the Deliberator and the Sequencer layer communication. The score differential here is much smaller than those of only the Deliberator or Sequencer communication, and this is simply a result of all agents knowing where the ball is located. With this communication setup, each agent spends less time reacquiring the ball position and instead spends more time acting upon the information. The reason that the goal differential is still greater than three is that there is an additional aspect to HAMR, where the Coordinator layer generates a better distribution of marking assignments when on defense, thus allowing the team to recover the ball better and keep it on the opponent's side of the field. The Layered Multirobot Architecture implementation has no such coordination. This reduces the team's ability to recover the ball when on defense, and increases time of possession for HAMR, thereby contributing to the score differential.

In general, the tests involving HAMR and LMA are performed to show the benefit to performance achieved by controlling and managing the communication in a

central portion of the architecture. HAMR contains the central, integrated Coordinator layer whereas LMA contains the addition of communication on the architecture's layers. HAMR performs better than LMA in this domain, regardless of the configuration selected for the communication in LMA, illustrating this point.

The third test shows that, even though there is higher-level processing and more advanced coordination and behavior generation mechanisms in HAMR, any adverse affects to its performance are negligible. It performs as well as the Trilearn Base code while also providing a platform independence through the hierarchy that is not present in the Trilearn Base code. Thus, there is no loss to its performance, and the capability is actually expanded, since the advancements provide capabilities not available in the Trilearn team's behavior-based approach. This capability expansion includes increased time of possession and better anticipation of ball position for intercepting the ball.

5.3 Summary

In the RCSS environment, the Layered Multirobot Architecture proves less effective than HAMR, as evidenced by the communication overhead and the score differential. The communication overhead caused by heavy message traffic in the Layered Multirobot Architecture results in a number of communication messages being blocked, thus reducing the agent's ability to operate efficiently. An inherent drawback of the Layered Multirobot Architecture is its Sequencer-layer communications, which causes cross-agent behavior inhibitions and occasional erratic behavior. This is the greatest contributor to the score differential, and thus the primary factor in the performance difference. This, coupled with the communication message blocking, forces the agent to spend much of its time alternatively activating various behavior sets or reacquiring ball information, since the ball information communication messages are often blocked. This takes away from the agent's ability to effectively operate in a dynamic environment such as the RCSS. It is not, however, any fault of the hierarchy that it encounters reduced performance. This is shown by the results of the testing between

HAMR and the Trilearn Base code. The hierarchical structure of HAMR causes no reduction in relative performance. The effectiveness of a single layer of communication versus communication on all layers is indicated by this, since the Coordinator provides a single centralized control for message generation and processing.

VI. Conclusion

Multi-Robot Systems (MRSs) used in certain situations provide a significant benefit, since the work is distributed among the robots in the group. In many applications, there remains a need for individual autonomy while also capturing the associated coordination aspects of the MRS. There is often a tradeoff between independence and coordination, since exclusive focus on either one leads to either a single-agent design or a highly cooperative yet mutually dependent multiagent design. Attempts to balance the two often result in either extraneous activity or error propagation. The design of the Hybrid Architecture for Multiple Robots (HAMR) emphasizes individual independence among members of a collective. The result of this independence is that HAMR provides greater robustness than some other architectural approaches, along with greater sophistication than most multirobot architectures since it is built on the three layered approach.

Comparison of HAMR against the Layered Multirobot Architecture in the RoboCup Soccer Simulator (RCSS) indicate that HAMR performs better than the Layered Multirobot Architecture in this application. The Layered Multirobot Architecture was selected as a basis for comparison since it is well suited to function in a single agent system, fully decentralized, and can make high-level decisions independently. The testing indicates that the addition of the Coordinator layer provides a greater benefit to a multi-agent system than additional communication on all layers. Specific benefits include minimizing interference by other agents, reducing communication overhead, greater modularity within an agent, and greater independence between agents. The results of the testing indicate that the Coordinator is a better way to bundle communication and manage coordinated activity, but only provides these benefits if it is an integrated part of the architectural design, as opposed to an addition to a preexisting architecture.

HAMR provides a number of key advancements:

1. provides coordination capabilities through an emphasis on individual autonomy.

2. contributes to coordination with low communication requirements.
3. provides a taskable system for all associated robots in the collective.
4. possesses a straightforward mechanism for modifying the collective size.
5. enables mutual independence for all the robots in the collective.

These were shown previously in the design of the architecture and in the evaluation of the architecture's performance in Chapters III and V.

6.1 Research Conclusions

The required properties for HAMR identified in Chapter I are to be lightweight in terms of communication, highly independent, highly cooperative, expandable, robust, and extensible. All these properties are fulfilled by HAMR, and are described below.

- Low communication overhead: all communications are conducted through the Coordinator. This provides a degree of filtering to reduce communication needs. In addition, the only required communications are those used to coordinate on tasks and assign utilities, so the demands on the communication system are low, especially when there is a low rate of task turnover.
- Highly independent: each agent in HAMR possesses high-level computation capabilities through a Deliberator, which translates goals into actions. In addition, any agent can complete a task when it falls within the agent's skillset and cooperation is not required.
- Cooperative: The Coordinator layer provides a mechanism for cooperating on tasks, allowing each agent to determine their own contribution to any task requiring cooperation.
- Expandable: addition of an agent to the group merely requires an addition of an address to the network, and other agents do not need to know about the

capabilities of the recently added agent. This provides a simple method to modify the group size easily and rapidly.

- Robust: any task that falls within the skillset of the remaining agents will be completed eventually, and the removal of one or more agents from the group does not cause group failure, since simple expiration times on tasks provides the ability to detect most failures.
- Extensible: the modularity of the three-layer architecture basis and the Unified Behavior Framework (UBF) provide a straightforward mechanism for introducing new skills to agents, and the other agents need not be informed of these changes.

The key advancements indicated previously are shown by the results of testing HAMR against the Layered Multirobot Architecture. HAMR provides coordination capabilities through an emphasis on individual autonomy with low communication requirements, shown by the defender assignments in the RoboCup Soccer Simulator (RCSS) team implementation of HAMR. It provides a taskable system for all associated robots in the collective, provided by the layered architectural approach and generation of the Play instance in the RCSS implementation. HAMR possesses a straightforward mechanism for modifying the collective size, since references to other players in the RCSS team are contained in a linked list, which is easy to modify. Finally, HAMR enables mutual independence for all the robots in the collective. This is shown by the nature of individual assignments and responsibilities in HAMR and formations and player types in the RCSS domain. Thus, HAMR provides a number of advancements over other architectures and provides these advancements by encouraging individual autonomy. The key to these advancements are in the Coordinator layer, which contributes to mutual independence by its mechanisms for centralized management of communication and contribution to auction-based task allocation. This independence leads towards greater expandability and robustness in MRSs with HAMR.

6.2 Future Work

There are auxiliary aspects to this work that serve to improve not only HAMR, but also prove applicable in other architectural approaches. Therefore, some of these aspects are presented below, along with a few that pertain almost exclusively to HAMR.

- Determine an appropriate cutoff for behavior complexity. Behaviors may range in implementation from an activation of a single motor in response to a stimulus to a highly sophisticated, complex function with its own internal planning and high-level computation capabilities. A standardized cutoff for this complexity would serve to increase modularity and provide a cleaner separation between layers.
- Design a common framework for determining candidate behaviors in the Sequencer. Barring the development of a Sequencer which has a canned response for all states, this problem is difficult without making the Sequencer hardware dependent. One possible approach is to have each behavior "provide" a function known to the Sequencer so that the Sequencer generates behavior sets containing behaviors that provide the needed functionality.
- Implement a high-level solver in the Deliberator. More than just partial order planning to break down a task into subtasks, the Deliberator in a multi-robot system must also determine task scheduling, determine task allocation, and perform task maintenance. There must also be consideration of common resources, which must also be appropriately allocated.
- Implement HAMR in a real-world, heterogeneous robotic team. The implementation of HAMR in the RoboCup Soccer Server (RCSS) avoids many problems common to real-world robotics, such as sensor noise and odometry error. The RCSS implementation is also simplified in that there are only ever two goals: gaining possession of the ball and scoring goals. This negates the need for a

sophisticated Deliberator, and, though the agents on the team have slightly different capabilities, all agents have the same simulated hardware. A real-world implementation provides a true test of HAMR's robustness.

- Create a manager for sensor and motor energy expenditure. This is needed in order to monitor and limit energy output, thereby avoiding death via an internal cause. If a set of behaviors only utilize a limited number of sensors and there is no estimated need for the others, it is best to deactivate those sensors in order to reduce energy use and prolong the robot's period of operation. An example of this is night vision capabilities during daytime operations. The night vision provides no additional information to the state, and thus is wasted energy.
- Sensor component scheduling. Similar to that mentioned above, this addresses the need to shift computational resources to certain sensors in times of high activity. For example, if moving rapidly through an environment, more computational resources should be shifted to sensors that contribute to obstacle detection and avoidance, since there is a high demand on these sensors. Also, in times of low activity, a scheduling mechanism must allocate appropriate time blocks to each sensor so that there is no starvation (where certain sensors are never processed) and each sensor is providing pertinent and timely data. This also works in conjunction with the above item, where the usage of an energy-intensive sensor must become more intermittent when energy expenditure is of concern.
- Develop a mechanism for changing common Deliberator and Sequencer processing results into behaviors. This is, in essence, learning new behaviors based upon common behavior set activations and results. A related example is with a child learning to throw a ball. Initially, the ball throwing task consists of multiple sequences of arm movement behaviors. Over time, however, the child learns the new behavior, no longer considering it a collection of behaviors. This also enables greater precision in that the new behavior allows the focus to shift to the target of the throw instead of the throw itself.

- Generate a mechanism for enabling the full spectrum of both the cooperative and collaborative aspects of coordination in independent agents. Independence readily contributes to collaboration and loosely coupled cooperation, but is difficult to maintain in the face of tightly coupled cooperation, where the requirements for inter-agent synchronization are high. Other architectures and approaches handle this by increasing the dependence of the agents upon each other, thus limiting the scalability of the MRS and the independence of the agents in the system. Tightly coupled cooperation in independent agents provides both of these features while enhancing the task fulfillment capability of the system as a whole.
- Create a means of dynamically generating goals associated with tasks. Control architectures are typically presented with a task that follows a strict composition protocol. The agents in the architecture fulfill this well-defined goal without the need for new goals and associated tasks due to these constraints. Dynamic goal generation and fulfillment provides decomposition well suited to dynamic environments and enables the completion of ill-defined goals. It also provides partial formulation of a plan and abstract, contigency-based planning for dynamic environments. An example of this is sending a military squad to clear a building when there is no knowledge of the interior layout. The squad dynamically generates goals on a per-room basis and handles contigencies appropriately.

These suggestions are by no means complete, but completion and integration of them into HAMR could provide a highly sophisticated robot which significantly expands the current limits on robotic control architectures.

6.3 Final Remarks

Continuing efforts to reduce the need for human input in robotic operations have greatly expanded the capabilities of modern robotic systems. As these systems become more pervasive, mechanisms must be in place to enable them to coordinate on tasks and operate as a cohesive unit. Real-world systems are expected to oper-

ate reliably and tasking of these systems are expected to grow, thus increasing the potential heterogeneity and size of MRSs. HAMR provides mechanisms for task coordination and does so with low communication needs. It also contributes to the coordination capabilities by providing a common point for group size modifications, task processing, and utility determination in the Coordinator layer. These contributions emphasize individual autonomy, thus enabling the robots to coordinate in a manner similar to humans. This provides advanced coordination without sacrificing individual autonomy and allows dynamic modification of the group size, ranging from one robot to the limit of the addresses available. This provides steps towards reducing the degree of human-in-the-loop needed for autonomous systems, namely MRSs. One commonly stated benefit of MRSs is that tasking can be maintained with reduced individual robot costs. HAMR takes another approach, emphasizing enhanced tasking as opposed to focusing on maintaining only that which could otherwise be achieved.

Appendix A. Unified Modeling Language Diagrams

This appendix contains the class diagrams indicating the internal structure of the components of the final architecture. These are presented in order to clarify the specifics of the implemented architecture as described in Chapter IV.

The Coordinator handles the inbound and outbound communication messages, calling the *GenMsg()* function which, in turn, calls one of the message generation functions (*sayOppAttackerStatus()* or *sayBallStatus()*) if necessary. The *sayOppAttackerStatus()* and *sayBallStatus()* functions generate the messages formatted just as described previously. The **Deliberator** class, shown in Figure A.2, contains the functionality of the Deliberator. It contains a reference to the Sequencer and the World Model. The Sequencer is referenced in order to provide accessiblity to the Sequencer for passing down a play once it is generated. The World Model reference is present to provide access to the state so the Deliberator can make better decisions. The major functionality of the Deliberator is in the *genPlay()* function. This generates an instance of the **Play** class, shown in Figure A.3. This instance is then passed down to the Sequencer and the Sequencer runs the play. The **Sequencer** class (Figure A.4) contains references to all the behaviors, the world model, an arbitration unit, and a composite behavior that it generates using behaviors and the arbitration unit reference. The behaviors are referenced by the superclass **Behavior** to take advantage of polymorphism. The primary functionality of the Sequencer is in the *runPlay()* command, which conditionally generates composite behaviors for use with the UBF then returns the resulting action. This is performed in the *runPlay()* function by examining the World Model, then creating an individual behavior hierarchy using the UBF with regards to the play currently active. The Deliberator calls the *SetPlay()* function to set the play. The UBF class diagram, shown in Figure A.5, Shows the interconnections between classes in the UBF. This follows the pattern shown in chapter III, and is generated by the Sequencer. This behavior hierarchy composes the Controller layer. Figure A.6 shows the **Player** class, which contains the Sequencer, Deliberator, and Coordinator and calls upon them when appropriate. The *mainLoop()* function

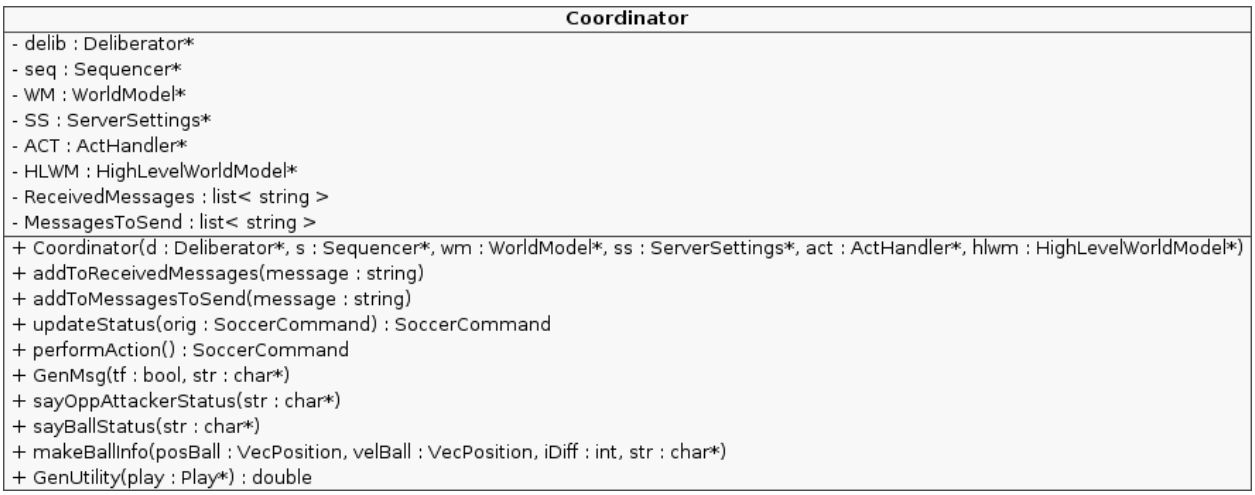

Figure A.1: The Coordinator Class Diagram.

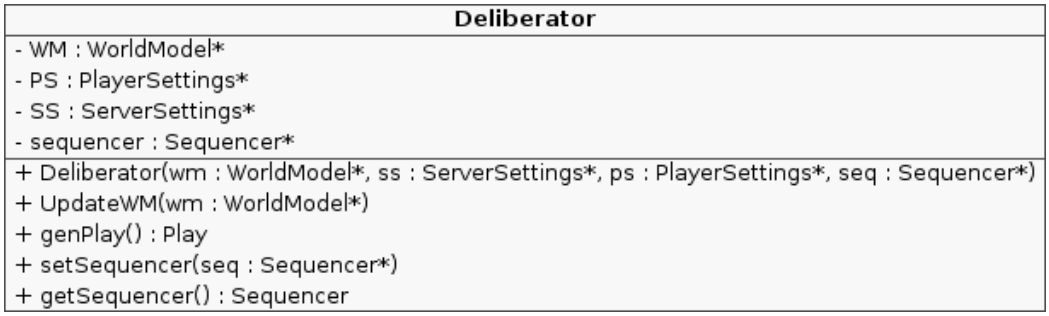

Figure A.2: The Deliberator Class Diagram.

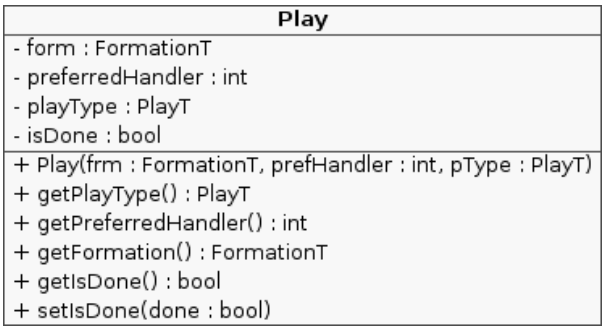

Figure A.3: The Play Class Diagram.

in the Player class calls the appropriate main loop for the player type, be it goalie, defender, midfielder, or attacker. These call the Sequencer's *runPlay()* function and generate an action, which is passed to the Server to execute.

Figure A.4: The Sequencer Class Diagram.

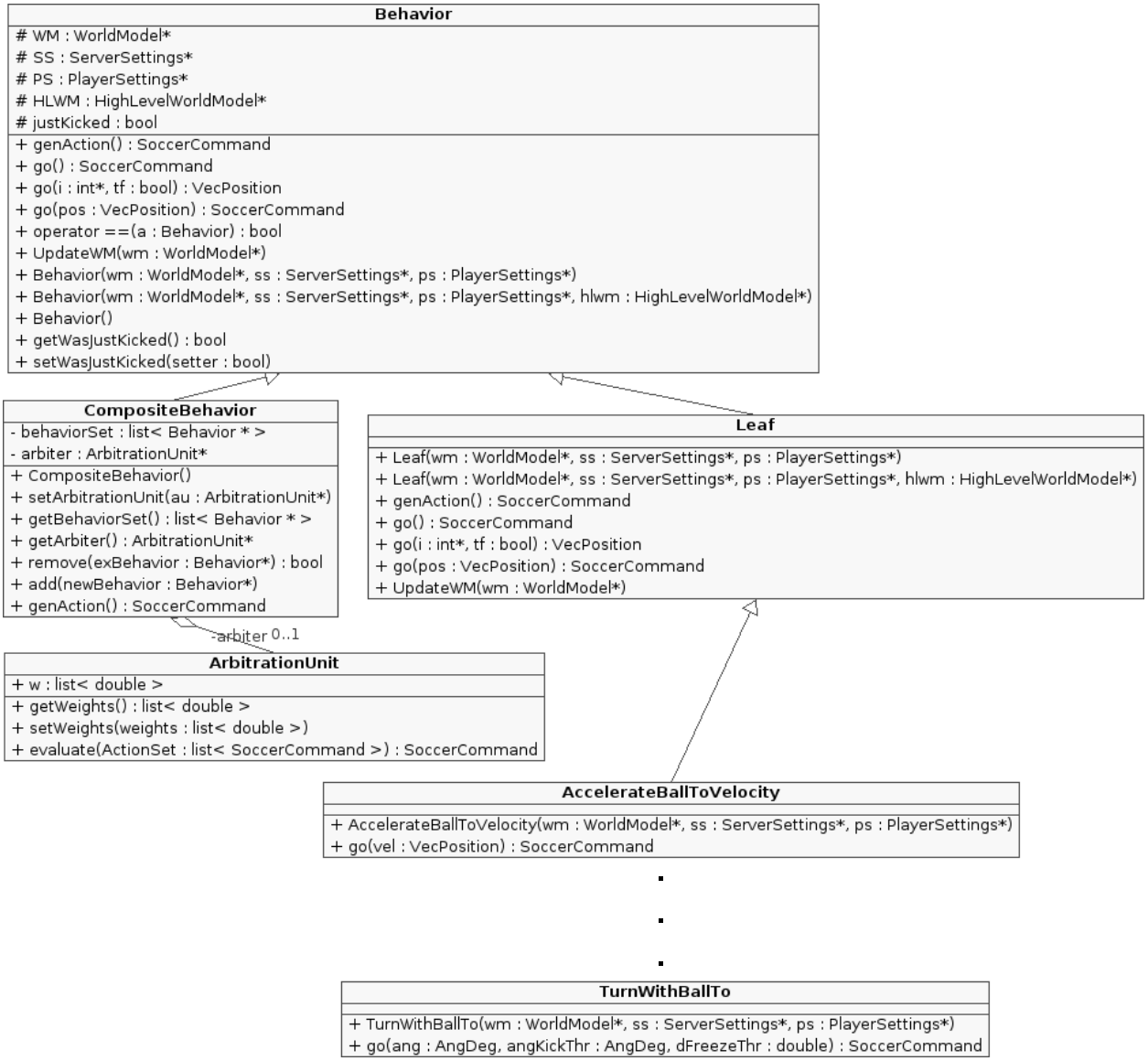

Figure A.5: The UBF class diagram, as implemented.

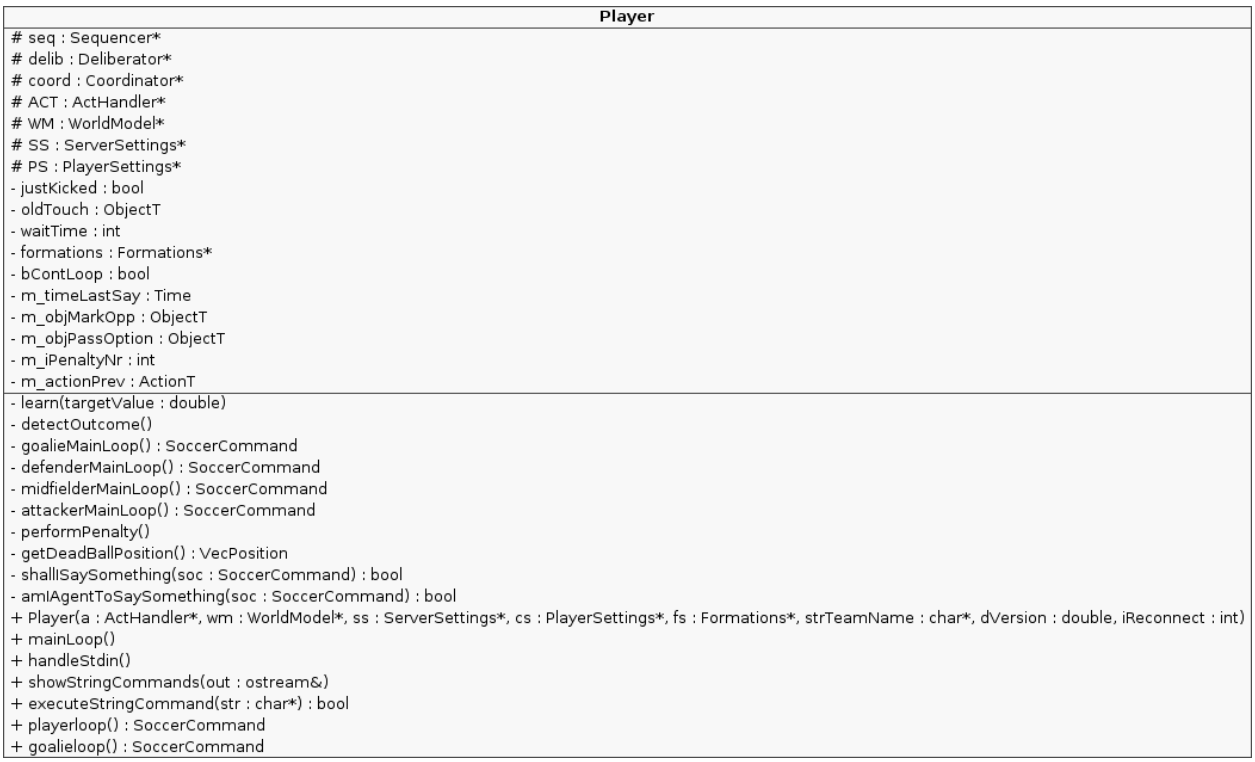

Figure A.6: The Player Class Diagram.

Bibliography

1. Arkin, Ronald C. "Behavior-Based Robot Navigation for Extended Domains". *Adaptive Behavior*, 1:201–225, 1992.

2. Arkin, Ronald C. *Behavior-Based Robotics*. MIT Press, Cambridge MA, 1998.

3. Arkin, Ronald C. and Tucker R. Balch. "AuRA: Principles and Practice in Review". *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 9:175–188, 1997.

4. Arkin, Ronald C. and Douglas MacKenzie. "Temporal Coordination of Perceptual Algorithms for Mobile Robot Navigation". *IEEE Transactions on Robotics and Automation*, 10(3):276–286, 1994.

5. Balch, Tucker and Ronald C. Arkin. "Communication in Reactive Multiagent Robotic Systems". *Autonomous Robots*, 1(1):27–52, 1995.

6. Balch, Tucker and Lynne E. Parker (editors). *Robot Teams: from Diversity to Polymorphism*. A K Peters, Ltd., 2002.

7. Bertsekas, Dimitri. "Auction Algorithms for Network Flow Problems: a Tutorial Introduction". *Computer Optimization Applications*, 1:7–66, 1992.

8. Blum, Avrim L. and John Langford. "Fast Planning Through Planning Graph Analysis". *Artificial Intelligence*, 90:281–300, 1997.

9. Bonasso, R. Peter, James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Marc G. Slack. "Experiences with an Architecture for Intelligent, Reactive Agents". *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):237–256, April 1997.

10. Bond, Alan H. and Les Gasser (editors). *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, 1998.

11. Brooks, Rodney A. "A Robust Layered Control System for a Mobile Robot". *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

12. Bryson, Joanna and Brendan McGonigle. "Agent Architecture as Object Oriented Design". *The Fourth International Workshop on Agent Theories, Architectres, and Languages (ATAL-97)*, 15–30. Springer-Verlag, Providence RI, 1998.

13. Camacho, David, Fernando Fernández, and Miguel A. Rodelgo. "Roboskeleton: An architecture for coordinating robot soccer agents". *Engineering Applications of Artificial Intelligence*, 19:179–188, 2006.

14. Cao, Y. Uny, Alex S. Fukunaga, and Andrew B. Kahng. "Cooperative Mobile Robotics: Antecedents and Directions". *Autonomous Robots*, 4(1):7–27, March 1997.

15. Chen, Mao, Ehsan Foroughi, Fredrik Heint, ZhanXiang Huahg, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Pat Rile, Timo Steffens, Yi Wang, and Xiang Yin. "Users Manual: RoboCup Soccer Server for Soccer Server Version 7.07 and later", August 2002.

16. Connell, Jonathan. "A Behavior-Based Arm Controller". *IEEE Transactions on Robotics and Automation*, 5:784–791, 1989.

17. Connell, Jonathan H. "SSS: A Hybrid Architecture Applied to Robot Navigation". *Proceedings of the 1992 IEEE Conference on Robotics and Automation*, 2719–2724. 1992.

18. Coušin, Kevin and Gilbert L. Peterson. "Distributed Approach To Solving Constrained Multiagent Task Scheduling Problems", November 2007. Proceedings of the 2007 AAAI Fall Symposium-Regarding the "Intelligence" in Distributed Intelligent Systems (RIDIS).

19. Dudek, Gregory, Michael Jenkin, and Evangelos Milios. *Robot Teams: from Diversity to Polymorphism*, chapter 1: A Taxonomy of Multirobot Systems. AK Peters, 2002.

20. Dudek, Gregory, Michael Jenkin, Evangelos Milios, and D. Wilkes. "A Taxonomy for Swarm Robots". *Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 441–447. Yokohama, Japan, 1993.

21. Fujita, Masahiro and Koji Kageyama. "An Open Architecture for Robot Entertainment". *AGENTS '97: Procedings of the First International Confernce on Autonomous Agents*, 435–442. Marina del Rey CA, 1997.

22. Georgeff, Michael P. and Amy L. Lansky. "Procedural Knowledge". *Proceedings of the IEEE*, volume 74, 1383–1398. October 1987.

23. Gerkey, Brian P. and Maja J. Matarić. "Sold! Auction Methods for Multirobot Coordination". *IEEE Transactions on Robotics and Automation*, 18(5):758–768, 2002.

24. Gerkey, Brian P. and Maja J. Mataric. "A Formal Analysis and Taxonomy of Task Allocation in Multi-Robot Systems". *The International Journal of Robotics Research*, 23(9):939–954, 2004.

25. Horswill, Ian. "A Laboratory Course in Behavior-Based Robotics". *IEEE Intelligent Systems*, 15(6):16–21, 2000.

26. Hunter, Matthew, Kostas Kostiadis, and Huosheng Hu. "A Behaviour-based Approach to Position Selection for Simulated Soccer Agents". *Proceedings of the 1st European RoboCup Workshop*. Amsterdam, May 2000.

27. Huntsberger, Terry, Paolo Pirjanian, Ashitey Trebi-Ollennu, Hari Das Nayar, Hrand Aghazarian, Anthony J. Ganino, and Mike Garrett. "CAMPOUT: A Control Architecture for Tightly Coupled Coordination of Multirobot Systems

for Planetary Surface Exploration". *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 33(5):550–558, September 2003.

28. Jones, Christopher V. *A Principled Design Methodology for Minimalist Multi-Robt System Controllers*. Ph.D. thesis, University of Southern California, August 2005.

29. Jung, David and Alexander Zelinsky. "An architecture for distributed cooperative planning in a behaviour-based multi-robot system". *Robotics and Autonomous Systems*, 26:149–174, 1999.

30. Kinny, David and Michael Georgeff. "Modelling and Design of Multi-Agent Systems". *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, 1–20. Springer-Verlag: Heidelerg, Germany, Budapest, Hungary, 1996.

31. Kok, Jelle R., Nikos Vlassis, and F.C.A. Groen. "Uva Trilearn 2003 team description". *Proceedings CD RoboCup 2003*. Springer-Verlag, Padua, Italy, July 2003.

32. Konolige, Kurt and Karen Myers. "The Saphira Architecture for Autonomous Mobile Robots". *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, 211–242, 1998.

33. Kortenkamp, David, R. Peter Bonasso, and Robin Murphy (editors). *Artificial Intelligence and Mobile Robots*, chapter 8: On Three-Layer Architectures. AAAI, 1998.

34. Krothapalli, Naga K.C. and Abhijit V. Deshmukh. "Distributed Task Allocation in Multi-Agent Systems". *Proceedings of Eleventh Annual Industrial Engineering Research Conference (IERC) 2002*. Orlando, Florida, May 2002.

35. Lee, Jaeho, Marcus J. Huber, Patrick G. Kenny, and Edmund H. Durfee. "UM-PRS: An Implementatin of the Procedural Reasoning System for Multirobot Applications". *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)*, 842–849. Houston TX, 1994.

36. Liu, Hindong, Huosheng Hu, and Dongbing Gu. "A Hybrid Control Architecture for Autonomous Robotic Fish". *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 312–317. Beijing, China, 2006.

37. Muscettola, Nicola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. "Remote Agent: To Boldly Go Where No AI System Has Gone Before". *Artificial Intelligence*, 103(1-2):5–47, 1998.

38. Myers, Karen L. "A Procedural Knowledge Approach to Task-level Control". B. Drabble (editor), *Proceedings of the 3rd International Conference on Artificial Intelligece Planning Systems (AIPS-96)*, 158–165. AAAI Press, 1996.

39. Palomeras, Narcis, Marc Carreras, Pere Ridao, and Emili Hernandez. "Mission Control System for Dam Inspection with an AUV". *Proceedings of the 2006*

IEEE/RSJ International Conference on Intelligent Robots and Systems, 2551–2556. Beijing, China, 2006.

40. Parker, Lynne E. "On the Design of Behavior-Based Multi-Robot Teams". *Advanced Robotics*, 10(6):547–578, 1996.

41. Parker, Lynne E. "Toward the Automated Synthesis of Cooperative Mobile Robot Teams". *Proceedings of SPIE Mobile RobotsXIII*, volume 3525, 82–93. Boston MA, Nov 1998.

42. Parker, Lynne E. "ALLIANCE: An Architecture for Fault Tolerant, Cooperative Control of Heterogeneous Mobile Robots". *Neurocomputing*, 28:75–92, 1999.

43. Peck, Roxy, Chris Olsen, and Jay Devore. *Introduction to Statistics and Data Analysis*. Duxbury, 2001.

44. Peterson, Gilbert L. and Diane J. Cook. "Incorporating Decision-Theoretic Planning in a Robot Architecture". *Robotics and Autonomous Systems*, 20(3):89–109, February 2003.

45. Rosen, Charles A. and Nils J. Nilsson. *Application of Intelligent Automata to Reconnaissance*. Technical report, Stanford Research Institute, March 1967.

46. Rosenblatt, Julio K. "Utility Fusion: Map-Based Planning in a Behavior-Based System". *Proceedings of FSR '97: International Conference on Field and Service Robotics*, 411–418. Canberra, Australia, 1997.

47. Russell, Stuart J. and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., 1995.

48. Simmons, Reid, Sanjiv Singh, David Hershberger, Josue Ramos, and Trey Smith. "First Results in the Coordination of Heterogeneous Robots for Large-Scale Assembly". *Proceedings of the International Symposium on Experimental Robotics(ISER)*. Honolulu HI, 2000.

49. Simmons, Reid G. "Structured Control for Autonomous Robots". *IEEE transactions on Robotics and Automation*, 10(1):34–43, February 1994.

50. Snedecor, George W. and William G. Cochran. *Statistical Methods*. Iowa State University Press, 8th edition, 1989.

51. Tang, Hongru, Aiguo Song, and Xiaobing Zhang. "Hybrid Behavior Coordination Mechanism for Navigation of Reconnaissance Robot". *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1773–1778. Beijing, China, 2006.

52. Utz, Hans, Stefan Sablatnög, Stefan Enderle, and Gerhard Kraetzschmar. "MIRO-Middleware for Mobile Robot Applications". *IEEE Transactions on Robotics and Automation*, 18(4):493–497, August 2004.

53. Verma, Deepak and Rajesh P.N. Rao. "Planning and Acting in Uncertain Environments using Probabilistic Inference". *Proceedings of the 2006 IEEE/RSJ*

International Conference on Intelligent Robots and Systems, 2382–2387. Beijing, China, 2006.

54. Volpe, Richard, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. "The CLARAty Architecture for Robotic Autonomy". *Proceedings of the 2001 IEEE Aerospace Conference*, volume 1, 121–132. Big Sky MT, March 2001.

55. Wooldridge, Michael, Nicholas R. Jennings, and David Kinny. "The Gaia Methodolog for Agent-Oriented Analysis and Design". *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

56. Woolley, Brian G. and Gilbert L. Peterson. "Unified Behavior Framework for Reactive Robot Control". Submitted for publication in ACM Transactions in Autonomous and Adaptive Systems.

Vita

Daylond J. Hooper graduated from Fairborn High School in Fairborn, Ohio. He enlisted in the Army in July 1998. His first assignment was at Fort Bragg, North Carolina, where he was assigned to A Company, 2nd Battalion, 505th Parachute Infantry Regiment of the 82nd Airborne Division. He deployed to Albania and Kosovo in April 1999 for six months. He received an honorable discharge in July 2000. In August 2002, he enrolled in undegraduate studies at Wright State University in Dayton, Ohio where he graduated Magna Cum Laude with a Bachelor of Science degree in Biomedical Engineering in June 2006.

He received the DAGSI fellowship for Ph.D. study and enrolled in the Graduate School of Engineering and Management, Air Force Institute of Technology in June 2006. Upon graduation from the Master's degree program, he will continue to pursue studies towards the Doctor of Philosophy degree.

Permanent address: 2950 Hobson Way
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
14–12–2007	Master's Thesis	June 2006 — Dec 2007

4. TITLE AND SUBTITLE

A Hybrid Multi-Robot Control Architecture

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Daylond James Hooper

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/08-02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Jacob Campbell, Civ
AFRL/RNYN
2241 Avionicx Circle
Wright-Patterson Air Force Base, OH 45433
(937) 255-6127 x4154; jacob.campbell@wpafb.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Multi-robot systems provide system redundancy and enhanced capability versus single robot systems. Implementations of these systems are varied, each with specific design approaches geared towards an application domain. Some traditional single robot control architectures have been expanded for multi-robot systems, but these expansions predominantly focus on the addition of communication capabilities. Both design approaches are application specific and limit the generalizability of the system. This work presents a redesign of a common single robot architecture in order to provide a more sophisticated multi-robot system. The single robot architecture chosen for application is the Three Layer Architecture (TLA). The primary strength of TLA is in the ability to perform both reactive and deliberative decision making, enabling the robot to be both sophisticated and perform well in stochastic environments. The redesign of this architecture includes incorporation of the Unified Behavior Framework (UBF) into the controller layer and an addition of a sequencer-like layer (called a Coordinator) to accommodate the multi-robot system. These combine to provide a robust, independent, and taskable individual architecture along with improved cooperation and collaboration capabilities, in turn reducing communication overhead versus many traditional approaches. This multi-robot systems architecture is demonstrated on the RoboCup Soccer Simulator showing its ability to perform well in a dynamic environment where communication constraints are high.

15. SUBJECT TERMS

control systems, robotics, cooperation, multiagent systems, control architectures

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Gilbert Peterson, Ph.D. (ENG)
U	U	U	UU	116	19b. TELEPHONE NUMBER *(include area code)* (937) 255–3636, x4314; gilbert.peterson@afit.edu